Habilitationsschrift

Grundlagen und Einsatzgebiete intelligenter Vorausschau

(Zusammenfassung)

Ulf Lorenz
Institut für Informatik
Paderborn

24. Februar 2006

Inhaltsverzeichnis

1 Persönliche Angaben

Dr. rer. nat. Ulf Lorenz, geb. 21.04.1969 in Gütersloh/Westfalen, verheiratet, 3 Kinder

2 Berufliche und akademische Laufbahn

Wissenschaftlicher Assistent, seit 2001
Institut für Informatik in Paderborn
Mitglied in der Arbeitsgruppe *Effiziente Nutzung paralleler und verteilter Systeme* unter Leitung von Prof. Dr. Burkhard Monien

Promotion im Fach Informatik, März 2001
Fachbereich Mathematik/Informatik, Universität-GH Paderborn
Thema der Dissertation: Controlled Conspiracy Number Search
Betreuer: Prof. Dr. Burkhard Monien
Gutachter/Kommission: Prof. Dr. I. Althöfer, Prof. Dr. S. Böttcher, Prof. Dr. B. Monien, Dr. P. Pfahler, Prof. Dr. F. Rammig
Abschluss: Dr. rer. nat. (summa cum laude)

Wissenschaftlicher Mitarbeiter, Dezember 1995 – März 2001
Fachbereich Mathematik/Informatik, Universität-GH Paderborn
Mitglied in der Arbeitsgruppe *Effiziente Nutzung paralleler und verteilter Systeme* unter Leitung von Prof. Dr. Burkhard Monien

Studium der Informatik, Oktober 1989 – Dezember 1995
Universität GH Paderborn
Abschluss: Diplom-Informatiker

Wehrpflichtiger (15 Monate), Feldjäger Kiel, 1988 - 1989

Allgemeine Hochschulreife, Einstein-Gymnasium in Rheda, 1988

3 Einleitung

In meinen wissenschaftlichen Arbeiten beschäftige ich mich mit verschiedenen Aspekten intelligenter Vorausschau bei Optimierungsproblemen und in PSPACE-schweren Spielen. Der Schwerpunkt meiner Arbeit liegt dabei auf so genannter Spielbaumsuche, die wohl am einfachsten folgendermaßen charakterisiert werden kann. Eine Person, die eine Entscheidung zu fällen hat, fragt sich in der ihr gegebenen Situation, welche Hier-und-jetzt-Entscheidungsoptionen sie hat, welche Ereignisse die Umgebung nach ihrer Aktionsentscheidung erzeugen kann, welche Optionen sie selber daraufhin wieder hat usw. Ob und unter welchen Bedingungen es sinnvoll ist, den entstehenden Baum möglicher Entwicklungen von einem Computer auswerten zu lassen, hängt von verschiedenen Aspekten wie der Problemkomplexität und Approximierbarkeit, aber auch von instanzspezifischen Gegebenheiten ab.
Einerseits lege ich bei meiner Arbeit großen Wert auf das Erfassen und Verstehen fundamentaler Zusammenhänge und darauf aufbauend auf die Entwicklung neuer Ideen und algorithmischer Konzepte zur Lösung grundlegender Probleme. So habe ich Suchalgorithmen entwickelt und analysiert, sowie theoretische Analysen zur Fehlerfortpflanzung in Spielbäumen durchgeführt.
Andererseits war der Ausgangspunkt meiner wissenschaftlichen Tätigkeiten oft eine Problemstellung der Praxis. Ein weithin sichtbarer Erfolg ist das Schachprogramm Hydra, das im Rahmen einer Industriekooperation entstanden ist. Meine Erfahrungen mit Suchalgorithmen, parallelen Algorithmen und verteilten Systemen und deren Programmierung haben entscheidend zum Erfolg des Programms beigetragen.
Die Methoden und Ideen, die ich im Bereich von Spielbaumsuchen entwickelt habe, lassen sich auf andere Bereiche wie z.B. Flugplanung übertragen. Die Entwicklung des so genannten Reparaturspiels für mehrstufige stochastische Planungsprobleme ist das Ergebnis einer Zusammenarbeit mit Lufthansa Systems. Hierbei wurde das Stochastische Flottenzuweisungsproblem herausgearbeitet und untersucht. Ziel des Spiels ist es, weniger störanfällige Pläne und Umplanungsstrategien zur Verfügung zu stellen. Die Hauptleistung liegt hier m.E. in der Formulierung des zugrunde liegenden Spiels, welches die Summe unserer Erkenntnisse über Komplexität, stochastische ganzzahlige mehrstufige Optimierung, mathematische Optimierungsmodelle und Planung sowie Spielbaumsuche mit Heuristiken darstellt.

In den zwei folgenden Unterkapiteln wird der Begriff des „Spiels“ präzisiert, und es wird der Zusammenhang zwischen Spielen, der Komplexitätsklasse PSPACE und Entscheidungsfindung unter Unsicherheit hergestellt. Kapitel 4 enthält die inhaltliche Zusammenfassung meiner Veröffentlichungen [K1-K16, Z1-Z6]. In Kapitel 4.1 werden neue Ergebnisse zur Fehlerfortpflanzung in Spielbäumen vorgestellt. Die Analyse zeigt, unter welchen Bedingungen heuristische Spielbaumsuche in minmax-Bäumen, mit und ohne Zufallskomponenten, einen Erfolg versprechenden Lösungsansatz darstellt. Sie ist das verbindende Element zwischen den im Anschluss vorgestellten praktischen Anwendungen. Kapitel 4.2 beschäftigt sich mit Zweipersonen-Nullsummen-Spielen. Hier wird insbesondere auf Details des Schachprogramms Hydra eingegangen. In Kapitel 4.3 wird der Ansatz des so genannten Reparaturspiels vorgestellt, welches dazu dienen soll, stabile Umplanungsstrategien z.B. in der Flugplanung zu ermöglichen. Zum Schluss werden die eigenen Veröffentlichungen, für diese Arbeit relevante Fremdliteratur, eigene Lehrveranstaltungen sowie Projekte, in denen ich mitgearbeitet habe, aufgelistet.

3.1 Spiele

Wenn ich von „Spielen“ rede, verstehe ich darunter Optimierungsprobleme mit mehr als einem Akteur. Darunter fallen Ökonomische Spiele, Spiele gegen die Natur u.s.w. Mit dem Begriff „Spiel“ möchte ich mich in erster Linie von den Optimierungsproblemen abgrenzen, bei denen eine deterministische Eingabe ohne exogene Einflüsse zu einer optimalen Ausgabe verarbeitet werden soll. In der Literatur wird ein System aus Spielern, Aktionsregeln der Spieler und einer Funktion, die jedem Spieler am Ende einer Aktionssequenz seinen Gewinn zuweist, ebenfalls „Spiel“ genannt [25]. Im Zusammenhang mit Spielen, die mich interessieren, spielt die Komplexitätsklasse PSPACE eine besondere Rolle, da sie u.a. mit Hilfe einiger kombinatorischer Zweipersonen-Spiele charakterisiert werden kann [40]. Papadimitriou [30] zeigt, dass sich die Komplexitätsklasse PSPACE mit Hilfe von Entscheidungsproblemen unter Unsicherheit charakterisieren lässt. Solche Probleme sind typischerweise durch einen zeit-diskreten Zufallsprozess, dessen Parameter durch unsere Aktionen dynamisch beeinflusst werden kann, bestimmt. Unsere Aktionen basieren dabei auf dem aktuellen Zustand des Systems, und der nächste Zustand ist eine Zufallsvariable, dessen Verteilung von unserer Aktion abhängt. Das Ziel ist dann die Minimierung der erwarteten Kosten einer vorgegebenen Kostenfunktion über die Historie der Zustände und Aktionen. Auch das folgende sehr leicht verständliche Beispiel gehört dazu. Es handelt sich um das „Stochastische Satisfiability Problem“ (SSAT):

> Sei eine boolsche Formel F in konjunktiver Normalform und drei Literalen je Klausel gegeben. Sie bestehe aus Variablen $x_1,...,x_n$, wobei n gerade sei. Die Frage ist, ob es eine Wahl für x_1 gibt, so dass nach zufälliger Wahl von x_2 (jeweils mit Wahrscheinlichkeit 0,5 für die x_i mit geraden i) es eine Wahl für x_3 gibt u.s.w., so dass die Wahrscheinlichkeit, dass die Formel F wahr wird, größer als 0,5 ist. Papadimitriou beschreibt die Formeln folgendermaßen, wobei R als „für zufälliges“ gelesen wird:
>
> $$F = \exists x_1 R x_2 \exists x_3 ... R x_n [Pr(F(x_1,...,x_n)=true) > ½]$$
>
> Theorem: SSAT ist PSPACE-vollständig [30].

Offenbar kann man SSAT als Zweipersonen-Spiel zwischen einem Entscheider und dem Zufall auffassen. Außerdem zeigt das SSAT-Problem, dass die Planung unter Unsicherheit im Allgemeinen selbst dann PSPACE-schwer bleibt, wenn wir die in der stochastischen Optimierung übliche Vereinfachung vornehmen, dass die Natur unabhängig von unseren Aktionen agiert. Die Wahrscheinlichkeit, dass x_i auf wahr gesetzt werden wird, ist unabhängig von unserer Aktion an x_1, x_3, … x_{i-1}.

Statt eines vom Zufall gelenkten Gegenspielers könnten wir uns auch einen bösartigen Gegenspieler vorstellen, der seine x_i immer so setzt, dass die Restformel möglichst unerfüllbar wird. Wir erhalten damit das bekannte QSAT-Problem, welches meist als Standardvertreter der Klasse PSPACE angesehen wird.

Sofern die möglichen Aktions- Reaktionssequenzen eines Zweipersonen-Spiels polynomiell in ihrer Länge beschränkt sind, lassen sich die vorgestellten Zweipersonen-Spiele mittels einer Tiefensuche, wie z.B. dem Alphabeta-Algorithmus, mit polynomiell viel Platz lösen.

3.2 Entscheidungsfindung unter Unsicherheit

Spezielle, für meine Arbeit interessante Klassen von Spielen sind die folgenden:

Planungsaufgaben, in denen Unsicherheiten durch Wahrscheinlichkeiten modelliert werden

Wenn zwei Spieler nicht gegensätzliche Interessen haben, sondern einer der Spieler, wenn er am Zug ist, seine Aktionen zufällig aus einer Auswahl von möglichen Aktionen auswählt, erhalten wir eine stochastische Optimierungsaufgabe. Bei diesen Spielen besteht der implizit gegebene Spielbaum aus zwei Knotenarten: Max-Knoten, die Zustände repräsentieren, an denen der so genannte Max-Spieler am Zug ist und Avg-Knoten, die Zustände repräsentieren, bei denen der Avg-Spieler am Zug ist. An einem beliebigen Avg-Knoten v ist dessen Wert durch ein gewichtetes Mittel der Werte der Nachfolger von v bestimmt, an beliebigem Max-Knoten v durch das Maximum der Werte der Nachfolger von v. Beispiele sind das SSAT-Problem sowie das stochastische Fleetassignment-Problem, auf das ich in Kapitel 4.3 näher eingehe.
Flugplanungsprobleme sind sehr komplex und ihre effiziente Lösung oftmals eine sehr schwierige Aufgabe. Zunächst müssen die Probleme in formale mathematische Modelle transformiert werden, und selbst für die Lösung deterministischer Teilprobleme werden die fortgeschrittensten Algorithmen benötigt, um das hergeleitete Optimierungsproblem in den Griff bekommen zu können. Um einen Schritt weitergehen zu können und unser Wissen über Datenungenauigkeiten mit in den Planungsprozess einfließen lassen zu können, ist es meiner Ansicht nach unumgänglich, sowohl nach beweisbar guten Algorithmen in abstrakten Modellen zu suchen, als auch mit Hilfe experimenteller Studien zu neuen Erkenntnissen zu gelangen.

Nullsummenspiele

In Nullsummenspielen haben alle Spieler gegensätzliche Interessen, d.h., was ein Spieler hinzugewinnt, muss er einem anderen wegnehmen. Es gibt sie in verschiedenen Ausprägungen. Zum einen gibt es die so genannten Zweipersonen-Nullsummenspiele mit vollständiger Information, die oft mit Hilfe eines Spielbaums modelliert werden, der zwei Knotentypen enthält, Min- und Max-Knoten. Man geht davon aus, dass an den Blättern des Spielbaums die Spielausgänge bekannt sind. An einem beliebigen Max-Knoten v ist dessen Wert durch das Maximum der Werte der Nachfolger von v bestimmt, an beliebigem Min-Knoten v durch das Minimum der Werte der Nachfolger von v. Beispiele sind Gesellschaftsspiele wie Schach, Go, sowie das QSAT-Problem u.v.m. Wenn der Spielbaum so geartet ist, dass alle seine Blätter auf Tiefe t liegen und alle Knoten einen Verzweigungsgrad von b haben, untersucht der so genannte Alphabeta-Algorithmus zumindest im besten Fall lediglich $O(b^{t/2})$ viele der b^t Blätter, ohne das Ergebnis an der Wurzel zu verfälschen.

Wenn zusätzlich noch ein Würfel im Spiel ist, wie bei Backgammon, bestehen die Spielbäume aus drei verschiedenen Arten von Knoten: Min-, Max- und Avg-Knoten.

Sind mehr als zwei Spieler involviert, ist keine allgemein anerkannte gute Modellierung der Spieler bekannt. Nimmt man an, dass jeder Spieler seinen eigenen Nutzen maximieren möchte, kann man zwar einen Spielbaum aufbauen, der entsprechend der Anzahl der teilnehmenden Spieler viele verschiedene Knotentypen beinhaltet. Es ist aber dann nicht möglich, den Baumknoten Werte so zuzuordenen, dass die einzelnen Spieler Werte von Knoten gegeneinander abwägen können.

4 Überblick über die eigenen Veröffentlichungen

Eine von mir entwickelte Idee, die dem tieferen Verständnis von Spielbaumsuche dient ist

- **Die Analyse von Fehlerfortpflanzung in *Minmax-Spielbäumen:**
  Blattdisjunkte Strategien, die in *Minmax-Spielbäumen vorhanden sein können, bestimmen die Robustheit der *Minmax-Auswertungen mit heuristischen Blattbewertungen. Durchschnitt-bildende Knoten erzeugen zusätzliche Robustheit.

Darauf aufbauend entstanden ist eine

- **Transformation der Fehlerfortpflanzungsanalyse in ein Schwierigkeitsmaß für Spielzustände.**
  Mittels eines abgewandelten Conspiracy Number Search Algorithmus bringt man einen Gegner mit nicht-perfekter heuristischer Bewertungsfunktion in Minmax-Spielen in Schwierigkeiten, wenn man selber über perfekte Wertinformation verfügt, aber nichts über die Fehlerhaftigkeit des Gegners weiß.

Da unsere Welt nicht nur aus Problemen besteht, die sich in relativ kleine, hochabstrakte Modelle pressen lassen, setze ich mich gezielt mit komplexeren, intuitiv praxisnäheren Modellen auseinander und suche den Wettbewerb mit anderen Modellen und dafür entwickelten Methoden. Das Ziel ist es dabei, immer neue Algorithmen, Heuristiken und Modellierungen zu finden, sowie starke Algorithmen, Heuristiken und Modellierungen von schwachen zu unterscheiden. Der Einsatz neuester Technologien, wie die Entwicklung von FPGA-Hardware, wird dabei berücksichtigt. Zwei, wie ich meine, hochinteressante Anwendungen, die ich mir genauer angeschaut habe, sind:

- **Das Schachspiel**
  Der Einsatz neuer FPGA-Technologie und dynamischer Lastbalancierung machen das Computerschach-Programm Hydra zur stärksten Schacheinheit der Welt.

- **Stochastisches Fleet-Assignment in der Flugplanung**
  Mittels der Exploration möglicher zukünftiger Ereignisse lassen sich robustere Flugpläne erstellen.

4.1 Analyse von Fehlerfortpflanzung in Spielbäumen

Ein Spielbaum kann, wie bereits angedeutet, häufig zur Modellierung von Vorausschau benutzt werden. In einer gegebenen Situation ermittelt der sich am Zug befindliche Spieler, welche Züge er zur Verfügung hat, welche Züge seine Gegner daraufhin ausführen können, welche er selber daraufhin ausspielen kann u.s.w. Aus dem entstehenden Baum, dessen Tiefe die Anzahl der Schritte angibt, die in mögliche Zukünfte geschaut werden kann, versucht der Spieler Erkenntnisse über richtige Entscheidungen an der Ausgangsstellung zu ziehen. Ersetzten wir „Situation“ durch „Knoten“, „Spieler“ durch „Knotentyp“ und „Züge“ durch „Kanten“, bekommen wir graphentheoretische Definitionen von Spielbäumen.

G = (T, h) ist ein Spielbaum, wenn T = (V, K) ein gerichteter Baum und h : V → $\mathbb{Z}$ eine Funktion, die Knoten auf ganze Zahlen abbildet ist und $V=V_1 \cup \ldots \cup V_n$ in mindestens zwei disjunkte Teilmengen aufgeteilt werden kann, die verschiedene Spieler repräsentieren.

Eine interessante Beobachtung ist, dass wir für viele Zweipersonen-Nullsummen-Spiele mit oder ohne vollständige Information zwar nicht die wahren Werte aller Stellungen kennen und auch nicht in der Lage sind, mittels Spielbaumsuche das Spiel vollständig auszuwerten, dass wir aber für viele Spiele gute heuristische Bewertungsfunktionen kennen, die Spielstellungen aussagekräftige Zahlen zuordnen. Spielbaumsuche zur Spielstärkesteigerung läuft dann wie folgt ab. Zunächst wird ein Teilbaum des Gesamtspielbaums zur Evaluierung ausgewählt, wie Abb.1 zeigt.

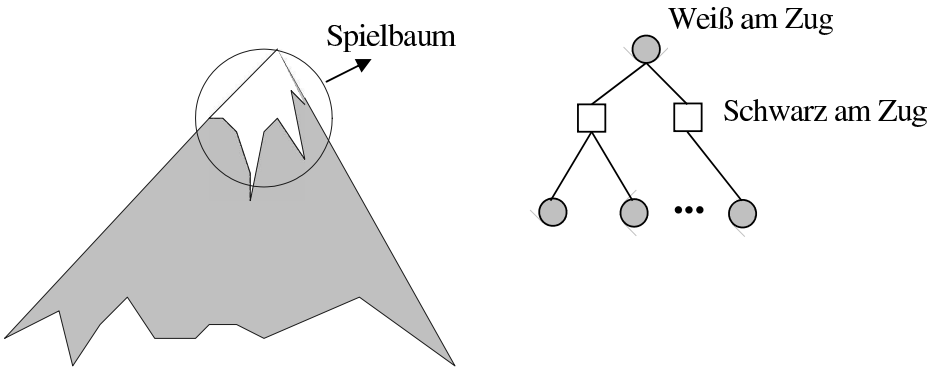


Abbildung 1. Nur der vorher ausgewählte Teilbaum eines Spielbaums wird ausgewertet

Die künstlich entstandenen Blätter werden mit Hilfe der heuristischen Bewertung bewertet und diese Werte werden zur Wurzel propagiert, als wären es die echten. Man spricht dann von heuristischer Spielbaumsuche. Die erstaunliche Beobachtung über die letzten 50 Jahre ist dabei, dass in Spielen wie Schach u.ä. der Spielbaum wie ein Fehlerfilter wirkt. Je größer dieser Filter, umso besser wird das Ergebnis!
Obwohl Spielbaumsuche ein wichtiges Thema der Artificial Intelligence darstellt und ganz speziell das Schachspiel sogar zur Drosophila der Artificial Intelligence erhoben [26] wurde, und obwohl gerade beim Beispiel Computerschach Erfolge und Fortschritte verzeichnet werden konnten, die man vor 50 Jahren nicht für realistisch gehalten hätte, gibt es bisher kein

allgemein anerkanntes Modell, das erklären kann, weshalb die heuristische Spielbaumsuche wie ein Fehlerfilter wirkt.

In den achtziger Jahren gab es sogar erste theoretische Analysen, die darauf hinwiesen, dass tieferes Suchen zu schlechteren Resultaten führen kann. Um dieses Phänomen zu untersuchen, gab es mehrere Arbeiten, u.a. von Pearl [31], Schrüfer [42], Althöfer [1], und Kaindl und Scheucher [18] und von Monien und mir [K6, Z2]. Die neueren der Arbeiten haben eine Idee gemeinsam: Die Autoren nehmen in ihren Modellen und Analysen an, dass es eine Wirklichkeit gibt, in der für alle Spielpositionen eindeutige, echte Werte existieren, die man aber nicht kennt. Die Kunst eines Spielprogramms, das nur heuristische Werte für Knoten zur Verfügung hat, ist es, so viel wie möglich über die echten Werte herauszufinden. Es wird dabei ferner angenommen, dass die heuristische Bewertungsprozedur der Spielprogramme entweder in dem Sinne sehr gut, aber nicht perfekt ist, dass der heuristische und der echte Wert eines Knotens hinreichend oft übereinstimmen, oder dass zumindest die Größer-Relationen zwischen Knotenpaaren hinreichend oft richtig eingeschätzt werden. Die Frage ist nun, unter welchen Bedingungen die Auswertung eines Spielbaums dabei hilft, die schwierige Aufgabe zu lösen, eine gute Entscheidung an der Wurzel des Suchbaums zu finden.

Spontane weitere Annahmen, die man jetzt zur Modellierung heranziehen könnte, sind z.B., dass sich das Verhältnis von Knoten, deren echte Werte man kennt, zu denen, die man nicht genau kennt, mit fortschreitender Baumsuche verändert oder dass die Qualität der heuristischen Werte mit zunehmender Suchtiefe zunimmt. Aus meiner Erfahrung mit dem Schachspiel scheinen mir diese Annahmen aber unrealistisch. Es ist keineswegs so, dass die Anzahl der Mattstellungen und Pattstellungen mit zunehmender Suchtiefe zunimmt. Im Gegenteil, nimmt man die Damen vom Brett, was in Endspielen häufig der Fall ist, nimmt die Anzahl der Mattstellungen im Suchbaum dramatisch ab. Auch dass die heuristische Bewertungsprozedur in Tiefe 18 von einer Startstellung ausgehend, bessere Werte als in Tiefe 16 liefern sollte, kann ich zumindest nicht für alle Spiele nachvollziehen.

Minmax-Spiele mit nur zwei Werten [Q2][K6][Z2]

Die Annahme, die wir in ein zunächst zweiwertiges Modell einfließen ließen, in dem es nur die Werte Gewinn und Verlust gibt, ist die, dass die Häufigkeit von Fehlern unabhängig von der Position im Baum sein soll. Die wichtigste Kernaussage unserer Analyse besagt dann, dass die Robustheit eines heuristischen Minmaxwertes nicht nur von der Anzahl der Gewinn- und Verluststellungen im Baum abhängt, sondern ganz stark von deren Anordnung. Von entscheidender Bedeutung ist dabei der Begriff der Strategie. Eine Strategie ist ein Teilbaum eines Spielbaums, der die Wurzel enthält, sowie für den einen Spieler immer genau einen Nachfolger von Strategieknoten und für den anderen Spieler immer alle Nachfolger der Strategieknoten. Strategien bestimmen in Minmax-Spielen untere und obere Schranken für den Minmax-Wert der Baumwurzel.

Betrachteten wir drei verschiedene Robustheitsmaße in Minmax-Spielbäumen mit den möglichen Werten 0 und 1. Das Ergebnis wird sein, dass diese drei Robustheitsmaße in gewissem Sinne equivalent zueinander sind und dass Robustheit nur dann gegeben ist, wenn der Spielbaum eine bestimmte Eigenschaft besitzt, dass er nämlich blattdisjunkte Strategien enthält. Im Detail sieht das so aus:

A) Sei ein beliebiger, aber fester Spielbaum G gegeben. Jedes seiner Blätter habe einen echten Wert 0 oder 1. Diese Werte sollen dem Minmax-Prinzip gehorchen. Wir weisen nun den Blättern zweite, so genannte heuristische Werte zu, die ebenfalls mit Hilfe des Minmax-Prinzips zur Wurzel propagiert werden. Mit Wahrscheinlichkeit p sollen der echte und der heuristische Wert von Blättern übereinstimmen. Bezeichne $Q_G(p)$ die Wahrscheinlichkeit, dass der echte und der heuristische Wert der Wurzel übereinstimmen. Wir bezeichnen nun die n-te Ableitung von $Q_G(.)$ mit $Q^n{}_G(.)$ und messen die Robustheit des Wurzelwertes mit Hilfe der ersten n Ableitungen von $Q_G(p)$ die 0 an der Stelle 1 sind. Diese Definition ist sehr intuitiv.

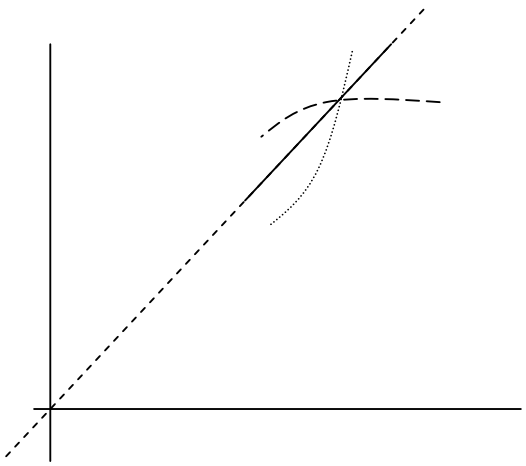


Abbildung 2: $Q_G(.)$ schneidet den Punkt (1,1)

Abb. 2 zeigt uns außer der Identitätsfunktion noch drei weitere mögliche Verläufe von $Q_G(p)$-Polynomen. Den Suchbaum G auszuwerten, macht offenbar nur dann Sinn, wenn $Q_G(p) > p$, da dann die Fehlerrate des Minmaxwertes der Baumwurzel geringer ist als die Fehlerrate einer Direktbewertung der Baumwurzel. Wenn es nun z.B. zwei Spielbäume G und H gibt, für die $Q^{(1)}{}_G(p) = 0$ und $Q^{(1)}{}_H(p) \geq 1$ gilt, so gibt es ein $\varepsilon>0$, so dass für alle $p \in [1-\varepsilon,1]$ $Q_G(p) > Q_H(p)$. Spielbäume, die beim so genannten iterativen Vertiefen untersucht werden, bilden eine Folge von Teil- bzw. Superbäumen. Die Forderung, dass zumindest ab einem hinreichend guten $p \in [1-\varepsilon,1]$ ein Superbaum ein besseres Fehlerverhalten als sein Teilbaum haben sollte, erscheint mir recht intuitiv, wenn es um die Frage geht, ob ein Spielbaum ein besserer Fehlerfilter ist als ein anderer.

B) Die Robustheit ist die Anzahl von Blättern, deren Wert man beliebig verändern kann, ohne dass dies Auswirkungen auf den Wert der Wurzel hat. Die Robustheit ist somit die Conspiracy Number eines Spielbaums und seines Wurzelwertes aus der Sicht echter Knotenwerte.
C) Die Robustheit eines Spielbaums wird definiert als die Anzahl blattdisjunkter Strategien, die entweder die obere Schranke 1 oder die untere Schranke 0 des Wurzelwertes zeigen.

Diese drei Robustheitsmaße sind equivalent zueinander. $Q^{(k-1)}{}_G(p) = ... = Q^{(1)}{}_G(p) = 0$ gilt genau dann, wenn es k blattdisjunkte Strategien in G gibt, die den Wurzelwert stützen, und die gibt es genau dann, wenn man die Werte von mindestens k-1 beliebig gewählten Blättern

beliebig ändern kann, ohne dass sich der Wurzelwert ändert. Insbesondere sichern k blattdisjunkte Strategien für kleines ε und konstantes c, dass $Q_G(1-\varepsilon) \geq 1-c\varepsilon^k$.

Erweiterung auf Spielbäume mit Min-, Max,- und Durchschnittknoten [Z6]

Um die obigen Ergebnisse auf Spielbäume mit Min- Max- und Durchschnittbildende Knotentypen zu erweitern und auch mehr als zwei mögliche Blattwerte berücksichtigen zu können, haben wir einige Definitionen, sowie unser Fehlermodell entsprechend erweitert. Ein Spielbaum G = ((V,E),h) kann nun die drei Knotentypen Min, Max und Avg enthalten, $h:V\rightarrow\mathbb{Z}$ soll die Bewertungsfunktion sein. Der so genannte *Minimax-Wert ist dann folgendermaßen definiert:

$$*\text{minimax}(v) = \begin{cases} h(v), \text{ falls v ein Blatt ist} \\ \max\{*\text{Minimax}(v') \mid (v,v') \in E\} \text{ falls v ein Max-Knoten} \\ \min\{*\text{Minimax}(v') \mid (v,v') \in E\} \text{ falls v ein Min-Knoten} \\ \text{average}\{*\text{Minimax}(v') \mid (v,v') \in E\} \text{ falls v ein Avg-Knoten} \end{cases}$$

Eine Strategie $S_s = (V_s,E_s)$ eines Spielers $s \in \{\text{Min, Max}\}$ in einem Spielbaum G = (V,E) mit Wurzel $r \in V$ ist nun ein Teilbaum von G, mit

- $r \in V_s$
- falls $u \in V_s$ ein innerer Knoten von G ist und u ein Knoten vom Typ s ist, gibt es genau einen Nachfolger u' von u mit $u' \in V_s$.
- falls $u \in V_s$ ein innerer Knoten von G ist und u ein Knoten vom Typ $\{\text{Min,Max}\}\backslash\{s\}$ ist, sind alle Nachfolger u in V_s.
- falls $u \in V_s$ ein innerer Knoten von G ist und u ein Knoten vom Typ Avg ist, gibt es eine nicht-leere Teilmenge C(u) der Nachfolgeknoten von v mit $C(u) \subseteq V$.

Wir wollen auch in diesem Modell darauf hinaus, dass Blätter mit einer nicht perfekten Heuristik bewertet werden, die den Blättern Zahlen zwischen 0 und 1 zuweist. Dafür führen wir einen Parameter $p \in [0,1]$ ein, der die Qualität der Heuristik beschreibt. Falls p = 1 ist, detektiert die Heuristik den echten Wert eines jeden Knotens sicher. Wenn p = 0 ist, soll die Heuristik keine verwertbare Information liefern: Wir nehmen daher an, dass es für jedes Blatt l eine feste, aber unbekannte Verteilung $p_l : [0,1] \rightarrow [0,1]$ gibt, so dass der Support

$$\text{supp}(p_l) := \{x \in [0,1] \mid p_l(x) \neq 0\}$$

endlich ist. Ferner sollen gelten $\{0,1\} \subseteq \text{supp}(p_l)$ und unsere Heuristik schätzt im Falle p = 0 den Wert von Blatt l als x mit der Wahrscheinlichkeit $p_l(x)$ ein. $\Sigma_{x \in \text{supp}(p_l)}$ ist natürlich gleich 1 und wir nehmen darüber hinaus noch an, dass $p_l(x) = 0$ ist, falls p = 0 und x der echte Wert von l ist. Falls der Qualitätsparameter p zwischen 0 und 1 liegt, wird zwischen den beiden

extremen Fällen (p=0 oder p=1) proportional zu p gewählt: Mit Wahrscheinlichkeit p detektiert die Heuristik den echten Wert von l, und mit Wahrscheinlichkeit (1-p)•p_l(x) gibt sie einen falschen Wert x ∈ [0,1] aus.

Die Wahrscheinlichkeit, dass der heuristische Wert eines Knotens v in G größer oder gleich einer Schranke γ ist, ist wiederum ein Polynom in p, das wir mit q(v,≥γ)(p) bezeichnen. $q^{(n)}$(v,≥γ)(p) bezeichne auch hier wieder die n-ten Ableitungen.
Auch in diesem Modell spielen blattdisjunkte Strategien eine Schlüsselrolle. Ist nämlich G mit Wurzel r ein Minmax-Spielbaum (also ohne Avg-Knoten), so ist genau dann $q^{(k-1)}$ (r,≥γ)(1) = ... = $q^{(1)}$ (r,≥γ)(1) = 0, wenn es k blattdisjunkte Strategien in G gibt, die alle belegen, dass der Wert r mindestens γ ist. Analoges gilt für Strategien, die obere Schranken für den Wurzelwert darstellen.

Wenn wir auch Avg-Knoten zulassen, gilt immer noch, dass aus der Existenz von k blattdisjunkten Strategien — die alle zeigen, dass der Wurzelwert größer oder gleich γ ist — folgt, dass $q^{(k-1)}$ (r,≥γ)(1) = ... = $q^{(1)}$ (r,≥γ)(1) = 0. Umgekehrt gilt dies nicht mehr, was bedeutet, dass Avg-Knoten zusätzliche Robustheit erzeugen können.

Direkte Anwendung [K10]

Natürlich gehören Gesellschaftsspiele, die selber eine Abstraktion, ein Modell der Wirklichkeit darstellen, auch in die Kategorie „Optimierungsproblem mit mehr als einem Akteur“. Da gibt es zum einen herausfordernde Spiele wie Schach oder Go, aber auch viele kleinere, die teilweise vollständig analysiert werden konnten, so dass man zu jeder Stellung den echten Minmaxwert kennt. Beispiele hierfür sind 4-Gewinnt, Mühle oder Endspieldatenbanken beim Schach. Erstaunlicherweise ist es gerade diese Art der Perfektion, die den Eindruck von Dummheit hinterlässt. Wenn nämlich ein ‚perfekt spielendes’ Programm eine Gewinnposition erreicht, wird es zwar die Partie immer gewinnen, und wenn es einmal eine Remisposition erreicht, wird es die Partie nicht verlieren. Wenn es aber gegen Gegner spielt, die kein perfektes Wissen über das gegebene Spiel besitzen, ist es trotzdem nicht unbedingt in der Lage, den Gegner in Schwierigkeiten zu bringen und im Laufe der Partie eine bessere Position zu erreichen als zu Beginn. Die existierenden Datenbank-basierten Programme für die genannten Spiele können nicht zwischen leichten und schwierigen Stellungen unterscheiden.

Mein Ziel war es, einen Algorithmus zu finden, der unter der Annahme dass, es eine Datenbank mit allen Stellungen und perfekten Wertinformationen gibt, perfekte Züge ausspielt und zugleich die Chancen dafür, dass der Gegner Fehler macht, maximiert. Nehmen wir zur Vereinfachung an, dass es, wie in den meisten Brettspielen üblich, drei Werte für Stellungen gibt und seien diese -1 für „Min-Spieler gewinnt“, 0 für „Remis“ und 1 für „Max-Spieler gewinnt“. Aus Symmetriegründen genügt es, sich Max-Spielbaumwurzeln anzusehen und das Schwierigkeitsmaß dann für Min-Knoten zu definieren. Zunächst ein Beispiel:

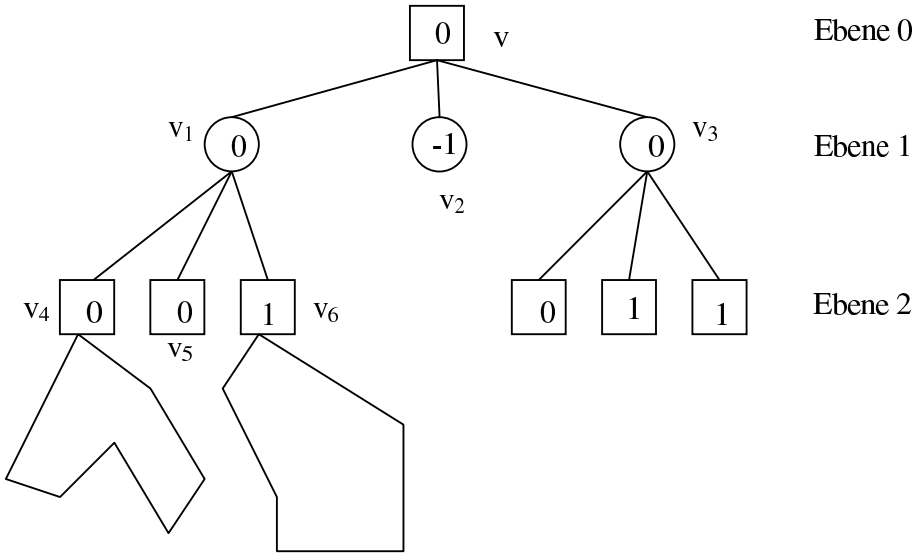


Abbildung 3: Baumausschnitt mit echten Minmax-Werten

In Ebene 0 der Abb. 3 geht es darum, einen möglichst guten Zug für den Max-Spieler zu finden, der sich dort in einer objektiv unentschiedenen Position befindet. (v,v_2) kommt als Zug nicht in Betracht, also bleiben die beiden Züge (v,v_1) und (v,v_3). Auf Ebene 1 ist der Gegner am Zug. Er wird versuchen, einen Zug auszuwählen, der ihm das Remis sichert, und er ist dabei beim Knoten v_1 in Gefahr, etwas falsch zu machen, wenn er entweder den Wert des Knotens v_6 unterschätzt oder wenn er die Werte der Knoten v_4 und v_5 überschätzt. Um zu einer für ihn guten Entscheidung zu gelangen, muss er also v_4 oder v_5 als guten Nachfolger von v_1 erkennen und zugleich sehen, dass v_6 ein schlechter Nachfolger von v_1 für ihn ist. Seien nun z_4 und z_5 die Anzahl von Knoten unter v_4 und v_5, die der jeweils kleinstmögliche Teilbaum mit zwei blattdisjunkten Strategien unter v_4 bzw. v_5 enthält. Die Strategien sollen natürlich belegen, dass der Wert von v_4 bzw. von v_5 kleiner oder gleich 0 ist. Sei z_6 die Anzahl Knoten des minimalen Baums unter v_6, der zwei blattdisjunkte Strategien dafür enthält, dass der Wert von v_6 größer oder gleich 1 ist. Dann definiere ich die „Schwierigkeit von v_1“ als $\min_{z4,z5} + z_6$. Die Idee ist, dass der Min-Spieler nicht genug Gewinn aus Spielbaumsuche ziehen kann, solange er nicht mindestens $\min_{z4,z5} + z_6$ viele Knoten untersucht, da es keinen signifikanten Fehlerfilter unter v_1 mit weniger Knoten gibt. Darüber hinaus kann der kleinste Teilbaum mit zwei blattdisjunkten Strategien beliebig irregulär sein, so dass der Min-Spieler auch noch die Schwierigkeit hat, seine Resourcen richtig einzusetzen.

Sei ein „Cn-C-Baum mit Wurzel v, der belegt, dass der echte Wert von v größer als (bzw. größer oder gleich, kleiner, kleiner oder gleich) X ist“ im folgenden ein Teilbaum des Spielbaums mit Min-Wurzel v, der die erwünschte Schranke X beweist und in dem man bis zu C-1 Blattwerte beliebig verändern kann, ohne dass die Schranke X an v verletzt wird. Die Funktion „difficulty“ sieht dann wie folgt aus:

Function difficulty(Min-Node v, robustness C)

// sei v ein Min-Knoten mit Wert -1 oder 0

// sei f die Funktion, die Knoten auf ihre echten Werte abbildet.

// seien $v_1,...,v_b$ die Nachfolger von v

if (f(v) == 0) {

cntSUM = $\Sigma_{f(vi)>0}$ Größe des kleinsten cn-C-Baums mit Wurzel v_i, der belegt, dass $f(v_i) > 0$.

cntMIN = minimum $_{f(vi) \leq 0}$ Größe des kleinsten cn-C-Baums mit Wurzel v_i, der belegt, dass $f(v_i) \leq 0$.

} else if (f(v) < 0) {

cntSUM = $\Sigma_{f(vi)\geq 0}$ Größe des kleinsten cn-C-Baums mit Wurzel v_i, der belegt, dass $f(v_i) \geq 0$.

cntMIN = minimum $_{f(vi)<0}$ Größe des kleinsten cn-C-Baums mit Wurzel v_i, der belegt, dass $f(v_i) < 0$.

}

Return cntMIN + cntSUM;

Ich habe dieses Schwierigkeitsmaß in einer intensiven experimentellen Analyse evaluiert. Die Experimente basieren auf über 1000 Partien, die acht verschiedene Programme in einem Rundenturnier gespielt haben. Dabei komme ich zu dem Schluss, dass die obige einfache und vom jeweiligen Spiel unabhängige Methode ein sehr mächtiges Werkzeug zur Bestimmung von Schwierigkeit einer Stellung ist. Zum ersten Mal wurde eine Methode angegeben, die ganz ohne Wissen über das jeweilige Spiel automatisch heuristische Stellungsbewertungen erzeugt, die ähnlich gut wie die von Schachspielern entwickelte ist.

4.2 Zweipersonen-Nullsummenspiele

Die Vorstellung, einen Computer entwickeln zu können, der in der Lage ist, ähnlich gut wie die jeweils besten Menschen intellektuelle Aufgaben meistern zu können, hat Forscher und Laien schon lange fasziniert. Ein berühmtes Beispiel ist die von Baron und Kempelen entwickelte Maschine „Der Schachtürke“, der scheinbar Schach spielen konnte. In Wirklichkeit handelte es sich um einen Jahrmarkttrick, und ein kleiner Schachspieler aus Fleisch und Blut war im Inneren verborgen.

4.2.1 Geschichte der Schachprogramme

Die ersten Schachprogramme, die ab den 1940er Jahren entwickelt wurden, versuchten recht erfolglos die menschlichen Spieler nachzuahmen. Erst in den 1970er Jahren demonstrierte dann das Schachprogramm Chess 4.5 [39], dass eine effiziente Spielbaumsuche ein fruchtbarerer Ansatz sein könnte. Belle [5], Cray Blitz [16], Hitech und Deep Thought [15] waren die Topprogramme der 80er Jahre, alles Programme, die auf besonders leistungsfähiger Hardware abliefen. Von 1992 an dominierten dann plötzlich die langsameren, aber clevereren PC-Programme die Computerschachwelt: ChessMachine, Fritz, Shredder etc. Die einzige Ausnahme bildet der historische Kampf zwischen Deep Blue [14] und Kasparov.

In der Zwischenzeit hat sich das Computerschach als eigene hochinnovative Disziplin entwickelt und es gibt mittlerweile fast alles, was das Herz begehrt: virtuelle Realität mit der Möglichkeit, übers Internet gegen Großmeister zu spielen, mit so genannten Maschinenräumen, in denen auch Computer mitspielen dürfen, mit der Möglichkeit, sein

eigenes Computerprogramm einzubinden und automatisch gegen andere Gegner spielen zu lassen uvm.
Nur eine große Herausforderung ist/war noch offen: Der endgültige und weithin anerkannte Triumph über die besten Mensch-Schachspieler. Was diesen Punkt angeht, schien es bis vor kurzem vier Kandidaten zu geben, namentlich Shredder, Fritz, Junior und Hydra. Diese vier Programme erzielen typischerweise 95% der Punkte gegen den Rest der Computerschachwelt, aber inzwischen scheint das Pendel zu Gunsten von Hydra ausgeschlagen zu haben.

Hydra ist das erste Schachprogramm, das auch die besten menschlichen Schachspieler klar schlägt und in Testpartien mehr als 3000 Elo sowie in einem wichtigen Wettkampf in London gegen den Großmeister Adams mehr als 3140 Elo-Punkte erzielte. Unsere Veröffentlichungen haben insofern besonderes Gewicht, als dass sie die Arbeitsweise des zur Zeit dieser Arbeit mit Abstand stärksten Schachprogramms vorstellen und darüber hinaus durch ihre Literaturverweise eine wichtige Auswahl aus den tausenden von Veröffentlichungen im Bereich des Computerschachs und selektiver Spielbaumsuche in Zweipersonenspielen vornehmen.

4.2.2 Das Schachprogramm Hydra [K16][K12][K9][Z5][Z4]

Wenn es darum geht, Computer Gesellschafts- und Brettspiele spielen zu lassen, bildet Spielbaumsuche seit vielen Jahren die algorithmische Grundlage, und die wichtigste Beobachtung über die letzten 40 Jahre hinweg in Spielen wie Schach und ähnlichen ist, *dass der Spielbaum in der Tat wie ein Fehlerfilter wirkt.* Deshalb gilt: Je schneller und je intelligenter der Suchalgorithmus ist, desto besser werden die Spielergebnisse!

Hydra ist ein Schachprogramm, in dem innovative neue Technologien und Techniken einen großen Teil der Spielstärke generieren. Die zwei auffälligsten Merkmale sind die Nutzung von Field Programmable Gate Arrays (FPGAs) sowie die Nutzung eines Cluster-Computers, auf dem der Alphabeta-Algorithmus effizient und verteilt abläuft.

Durch neue große Field Programmable Gate Arrays wird die Grenze zwischen Hardware und Software verwischt. Es ist nun möglich, komplexe Designs und feingranulare parallele Anwendungen ohne die langwierigen Chip-Design Zyklen von ASICs zu entwickeln.
Zugleich ist es durch Message-Passing Bibliotheken wie MPI deutlich einfacher geworden, grobgranulare parallele Anwendungen zu schreiben. Das Schachprogramm Hydra ist eine high-level Hardware/Software Co-Design Anwendung, die von diesen zwei Welten profitiert. Der zeitkritische Teil eines Suchbaums, nahe an seinen Blättern, wird mit Hilfe von feingranularer Parallelität von FPGA-Karten beschleunigt. Für Knoten nahe der Baumwurzel wird der Suchalgorithmus auf einem Cluster herkömmlicher Prozessoren verteilt.

Nach dem Vorbild von Deep Blue haben wir den Zuggenerator, die Bewertungsprozedur und den Suchalgorithmus in Hardware kodiert, was die folgenden Vorteile mit sich bringt: Zunächst können die Prozeduren in sehr wenigen Taktzyklen ausgeführt werden, was zu einer Steigerung der Geschwindigkeit führt. Ein weiterer wichtiger Punkt ist aber, dass es für PC-Programme einen Tradeoff zwischen Suchgeschwindigkeit und der Implementierung von Schachwissen gibt. Dieser Tradeoff existiert in Hardware (fast) nicht, da alle Bewertungsfeatures parallel ausgewertet werden können. Im Gegensatz zu ASIC-basierter Hardware haben FPGAs den Vorteil, dass Debugging wesentlich einfacher ist und dass Verbesserungen schnell hinzugefügt werden können.

4.2.2.1 Systemüberblick

Hydra nutzt das graphische Userinterface von ChessBase/Fritz, das auf einem gewöhnlichen WindowsXP-Rechner abläuft. Mit Hilfe einer secure shell verbindet sich dieses Interface über das Internet mit unserem Linux-Cluster, der wiederum aus 16 Dual-PC-Serverknoten besteht, die zwei PCI-Busse gleichzeitig kontrollieren können. Jeder PCI-Bus bekommt eine FPGA-Karte, und jeder MPI-Prozess wird auf einen der Prozessoren abgebildet und jedem dieser Prozesse ist eine FPGA-Karte zugeordnet. Die Serverknoten sind mit Hilfe eines Myrinet-Netzwerkes miteinander verbunden (Abb. 4).
Zur Zeit arbeiten wir mit XilinX basierten Virtex2Pro70 (xc2vp70) FPGA-Karten von Alphadata. Wir verwerten 127 von 328 BlockRAMs, 9380 von 333084 Slices, 558 von 66176 Flip-Flops und 17642 von 66176 LUTs. Der längste Pfad im Design besteht aus 54 Logikebenen und wir können das Design mit 50MHz takten. Eine obere Schranke für die Anzahl der Taktzyklen pro Suchknoten ist neun.

Experimente wurden teilweise auf Hardware der Universität Paderborn und teilweise auf dem Hydracluster in Abu Dhabi durchgeführt. Die Prozessoren sind Pentium IV/3.0Ghz, das Betriebssystem ist RedHat Linux.

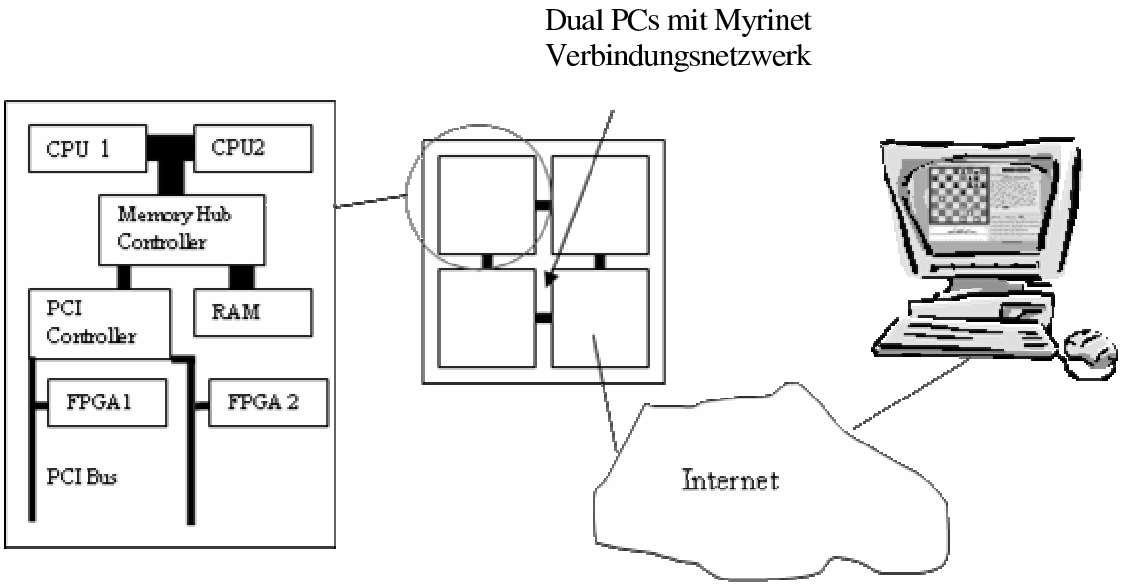


Abbildung 4: Hydras Architektur

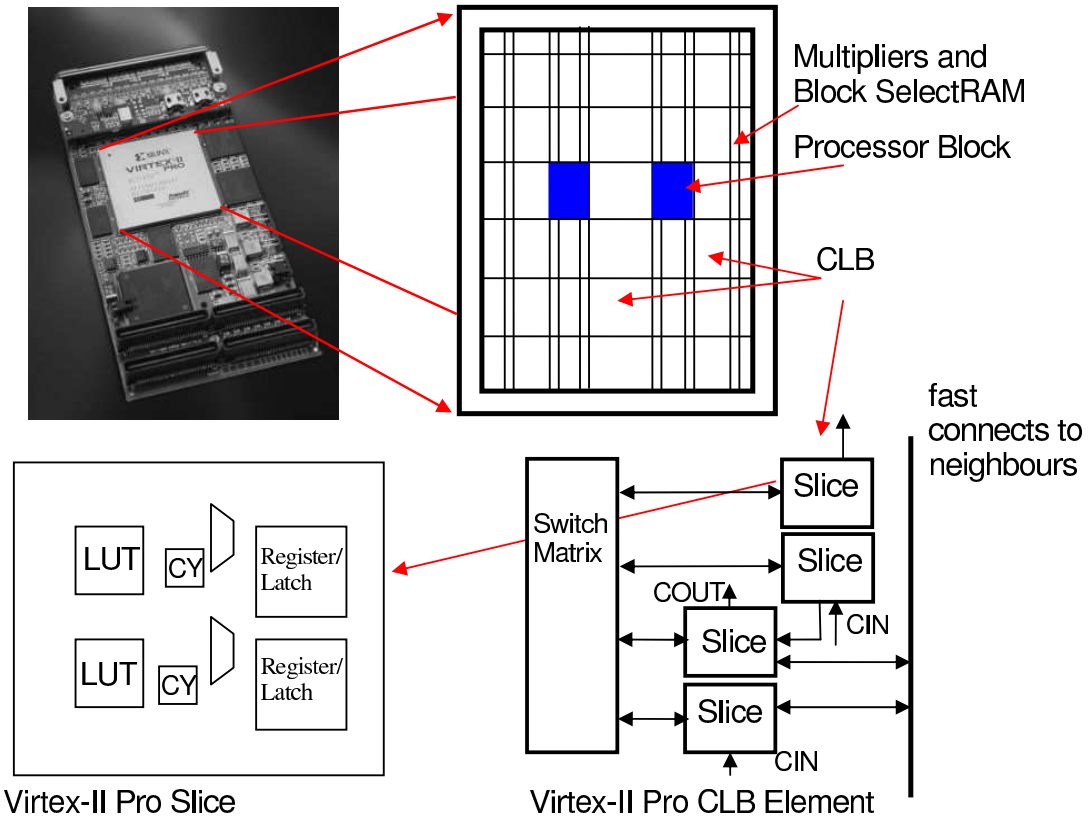


Abbildung 5: FPGA-Karte mit Xilinx-Chip

Abbildung 5 zeigt den prinzipiellen Aufbau eines Virtex-II Pro Chips, auf eine externe Karte montiert. Oben rechts der Abbildung sehen wir die generische Architektur des Virtex-II Pro mit seinen zwei PowerPC cores, Block RAM und den CLBs, die den eigentliche Kern eines frei konfigurierbaren Logikchips ausmachen. Ein CLB (Configurable Logic Block) besteht aus vier einander ähnlichen sogenannten Slices, die sehr schnelle Verbindungen zu ihren Nachbarn besitzen. Darüber hinaus ist jedes CLB mit einer Switchmatrix für globales Routing verbunden (s. Abb. 5, rechts unten). Jedes Slice (Abb. 5, links unten) enthält zwei 4-Input Funktionsgeneratoren, die als 4-Input Look-up Tables (LUTs) implementiert sind. Jedes LUT kann auch als 16-Bit RAM-Zelle oder als 16-Bit-Shiftregister benutzt werden. De facto ist eine LUT im Wesentlichen eine 16-Bit RAM-Zelle.

4.2.2.2 Baumsuche für intelligente Vorausschau

Die Suchprozedur von Hydra läuft in Teilen auf PCs und in Teilen auf FPGA-Karten. Weil der Flaschenhals des Systems der PCI-Bus ist, können wir auf der einen Seite nicht die gesamte Suche auf der PC-Seite ablaufen lassen. Auf der anderen Seite wäre es aber auch nicht klug, die gesamte Suche auf die FPGA-Karte zu verlagern, weil es in Hardware weitaus schwieriger ist, einen ausgeklügelten Suchalgorithmus zu implementieren. Wir teilen die Suche so auf, dass die letzten vier Halbzüge einer n-Halbzugsuche auf FPGA ablaufen,

inklusive der Ruhesuche und allen Bewertungen. Solch eine Tiefe-vier-Suche wird ca. 50000 mal pro Sekunde pro Prozessor angestoßen. Die Synchronisation zwischen der Suche auf der PC-Seite und der Suche auf der FPGA-Seite sieht folgendermaßen aus: Zu Beginn einer Baumsuche wird jeder Prozessor und auch jede FPGA-Karte über die aktuelle Stellung informiert. Dann werden alle Züge, die auf der PC-Seite ausgeführt oder zurückgenommen werden, an die FPGA-Karte inkrementell übermittelt. Wird ein Arbeitspaket an einen anderen Prozessor gegeben, läuft der Arbeitnehmerprozessor von der Startstellung bis zur übermittelten Baumstellung. Der Arbeitnehmerprozessor benachrichtigt seine FPGA-Karte entsprechend.

4.2.2.3 Der parallele Suchalgorithmus

Eine ganz entscheidende Kernkomponente des Schachprogramms Hydra ist ein für Clusterrechner parallelisierter Negascout-Algorithmus, eine Variante des Alphabeta-Algorithmus. Wir erreichen dadurch mit 32 Prozessoren eine Beschleunigung von 17,8, und Messungen in Form von Testspielen zeigen eine Spielstärkesteigerung von ungefähr 250 Elo-Punkten an. Bemerkenswert ist dabei, dass wir diese Beschleunigung erreichen, obwohl unsere Baumsuche Nullmoves [7] und Multicuts [3] verwendet. Beides sind Baumbeschneidungsheuristiken, die den Suchbaum sehr schmal werden lassen.

Die Hauptidee unserer Parallelisierung ist es, den Suchbaum zu zerstückeln [9][K9], mehrere Teile des Suchbaums gleichzeitig auszuwerten und die entstehende Last dynamisch mittels des Workstealing-Konzepts auf die Rechner des Netzwerks zu verteilen. Zunächst bekommt dabei ein Prozessor P_0 das Suchproblem und beginnt, dieses mittels des Negascout-Algorithmus sequentiell abzuarbeiten, als wäre er ganz allein. Gleichzeitig senden die anderen Prozessoren an zufällig ausgewählte Rechner im Netz Anfragen nach Arbeit. Wenn ein Prozessor P_i, der bereits Arbeit bekommen hat, solch eine Bitte einfängt, sieht er nach, ob es noch nicht untersuchte Teile seines eigenen Suchproblems gibt, die er zur parallelen Auswertung abgeben kann. Diese potentiellen Such-Teilprobleme liegen als rechte Geschwister von Suchknoten auf P_i's Suchstack. P_i sendet nun entweder solch einen Knoten (eine Schachposition mit Bounds etc.) zum anfragenden Prozessor P_j zurück, oder er sendet eine Absage. Falls ein Arbeitspaket versendet wurde, wird P_i zu einem Arbeitgeber und P_j zu seinem Arbeitsnehmer. Prozessoren können gleichzeitig Arbeitnehmer und Arbeitgeber sein, und diese Beziehungen ändern sich dynamisch während der Berechnungen. Wenn P_j seine Aufgabe erledigt hat (möglicherweise mit der Hilfe anderer Prozessoren), sendet er eine Ergebnisnachricht an P_i. Die Arbeitgeber/Arbeitnehmer-Beziehung zwischen den beiden ist dann beendet und P_j ist wieder arbeitslos. Er beginnt dann wieder Bitten um Arbeit ins Netz zu schicken.

Wenn ein Prozessor P_i herausfindet, dass er ein falsches Fenster zu einem seiner Arbeitnehmer geschickt hat, schickt er eine sogenannte Fensternachricht hinterher. P_j hält seine Arbeit dann an und startet die alte Suche von neuem. Falls die Nachricht einen Cutoff enthält, hält P_j seine Tätigkeit sofort an.

Im Gegensatz zu Datenintensiven Anwendungen, wie sie z.B. im Multimediabereich [19] auftreten, ist der Erfolg der Lastbalancierung in unserer Anwendung fast ausschließlich von den Latenzzeiten des Rechnernetzwerks und nur wenig vom maximalen Durchsatz abhängig.

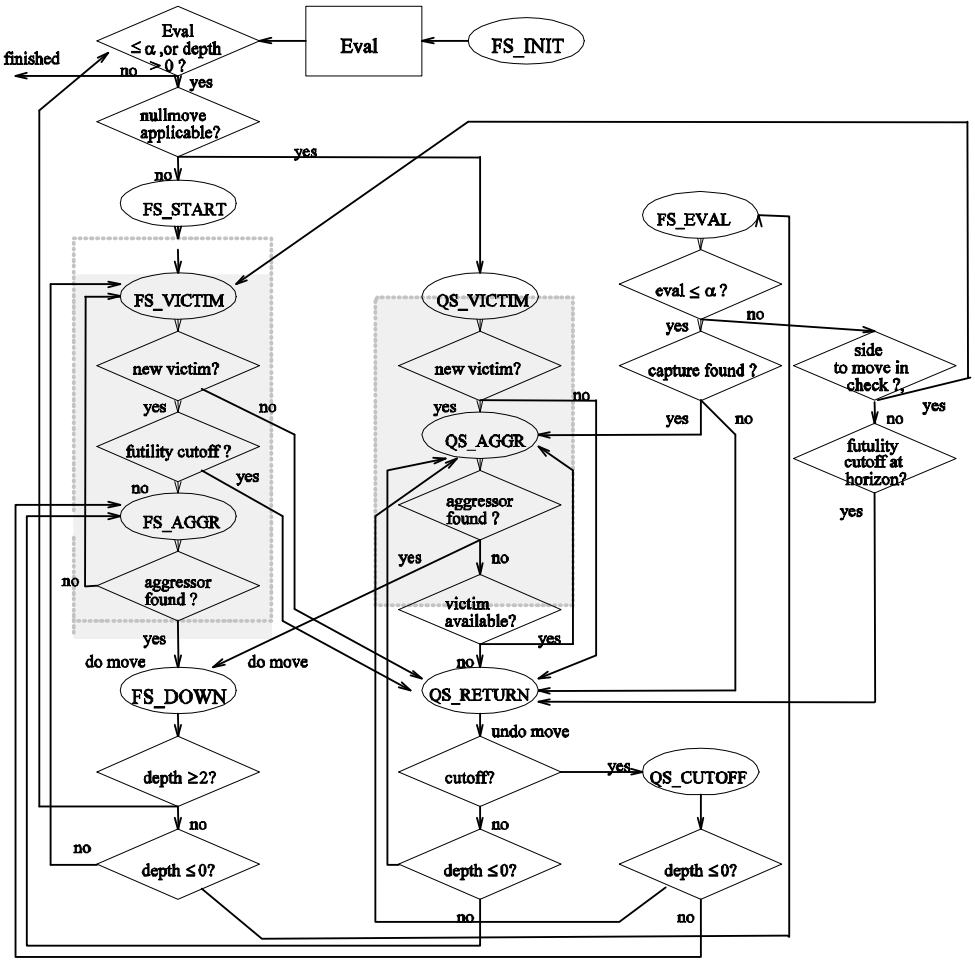


Abbildung 6: Vereinfachte Darstellung des endlichen Automaten mit 54 Zuständen**.**

4.2.2.4 Der Suchalgorithmus in Hardware [K9][K12]

Abb. 6 zeigt eine vereinfachte Version des endlichen Automaten (EA), der die Baumsuche auf der FPGA-Karte kontrolliert und die Signale von Zuggenerator und Bewertungsfunktion kontrolliert.

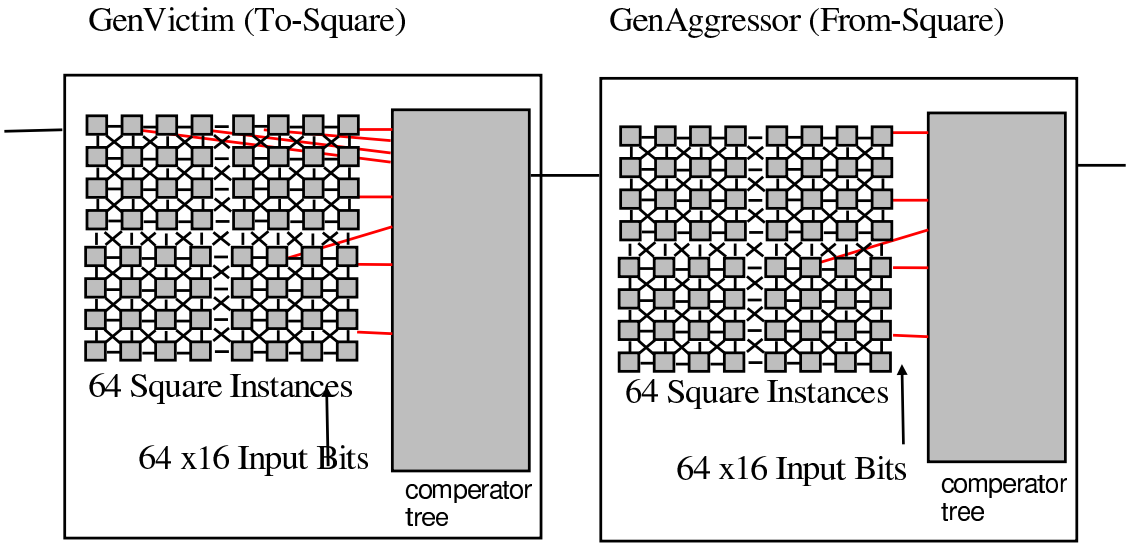
Die Suche funktioniert folgendermaßen: Wir betreten die Suche bei FS_INIT. Falls es überhaupt etwas zu tun gibt, und der Nullmove nicht angewendet werden kann, gelangen wir zur Hauptsuche. Nachdem möglicherweise die Restsuchtiefe erhöht wurde (nicht in Bild 4 dargestellt), geht der EA in den Zustand FS_VICTIM, in dem der Output des Moduls GenVictim inspiziert wird. Falls es ein sinnvolles nach-Feld gibt und kein sogenannter futility-cutoff möglich ist, erreichen wir den Zustand FS_AGGR, in dem der Output von GenAgressor inspiziert wird. Falls es ein geeignetes von-Feld zu dem gegebenen nach-Feld gibt, wird der Zug ausgeführt und wir erreichen den Zustand FS_DOWN. Dieser Zustand entspricht dem rekursiven Aufruf des Alphabeta-Algorithmus mit dem Suchfenster [alpha, alpha+1]. Falls nun die zu verbleibende Restsuchtiefe größer als Null ist, geht es zu FS_START, ansonsten betritt der EA den Teil der so genannten Ruhesuche, in dem nur noch Schlagzüge und einem Schach ausweichende Züge betrachtet werden. Die Ruhesuche beginnt mit der Untersuchung des Bewertungsmodul-Outputs. Falls die Bewertung nicht größer als alpha ist, fahren wir mit dem nächsten Schlagzug fort, sofern ein solcher vorhanden ist. Falls

eine Figur geschlagen werden kann, erreicht der EA den Zustand QS_AGGR, und falls zusätzlich ein sinnvolles von-Feld eines legalen Zuges von GenAggressor signalisiert wird, wird ein weiterer Zug ausgeführt. So geht es Suchknoten für Suchknoten weiter. Zurückgenommen werden Züge und Rekursionsebenen werden verlassen, wenn der EA den Zustand QS_RETURN erreicht.

Ein rekursiver Algorithmus wie der Alphabeta-Algorithmus benötigt einen Stack für sein Vorgehen. Der Stack wird in Hydra durch sechs Blöcke von Dualport BlockRAM realisiert. Das RAM ist als 16-bit RAM organisiert. Auf diese Weise können wir gleichzeitig entweder 2 16-bit Daten, oder 1 32-bit Wort in das RAM schreiben. Eine Tiefenvariable wird vom Such-EA kontrolliert, und verschiedene Tabellen sind für verschiedene lokale Variablen des Suchstacks zuständig.

4.2.2.5 Zuggenerator in Hardware:

Ein **Zuggenerator** wird in Software für gewöhnlich als Vierfachschleife implementiert. Eine Schleife über alle Figurentypen, eine innere Schleife für alle Figuren des einen Typs, eine weitere für die Richtungen, die so eine Figur einschlagen kann, und eine innerste Schleife, die diesen Richtungen so lange folgt, bis ein Hindernis auftritt. Das ist ein recht sequentielles Vorgehen, insbesondere wenn wir berücksichtigen, dass z.B. Schlagzüge an den Beginn der Zugliste sortiert werden sollten. In Hardware kann man einen schönen kleinen und schnellen Zuggenerator designen, der völlig anders funktioniert.



1. Occupied squares send a signal in GenVictim
2. Free squares forward these signals
3. All squares receiving a signal are potential to-squares
4. Comperator tree selects most attractive to-square (taking moves).

1. Winner square generates signal of a super piece
2. Free squares forward the signals
3. Squares occupied by own pieces are potential from-squares
4. Comperator tree selects most attractive from-square

Abbildung 7: Der Zuggenerator

Der Zuggenerator ist im Prinzip selber ein 8-mal-8 Schachbett (vgl. Abb. 7). Das sogenannte GenAggressor-Modul sowie das GenVictim-Modul instantiieren 64 Feldmodule, eines für jedes Feld. Beide bestimmen, zu welchen Nachbarn hereinkommende Signale weitergeleitet werden. Die Felder, auf denen Figuren stehen, senden Figursignale aus, bzw. leiten die hereinkommenden Signale weit reichender Figuren weiter. Zusätzlich kann jedes

Feld das Signal „victim found“ aussenden. Dann wissen wir, dass dieses Feld ein potentielles victim, d.h. ein Nachfeld eines legalen Zuges ist. Alle „victim found“-Signale gehen in einen Arbiter (eigentlich ein Komperatorbaum), welcher das attraktivste, bislang nicht untersuchte „victim“ auswählt. Das GenAggressor-Modul nimmt dann den Output des Arbiters als Input und sendet das Signal einer Superfigur (eine Kombination aller möglichen Figuren) aus. Falls nun z.B. ein Turmzug-Signal auf ein Feld trifft, auf dem ein eigener Turm steht, haben wir einen „Aggressor“, d.h. ein potentielles von-Feld eines legalen Zuges gefunden. Auf diese Weise werden viele Züge gleichzeitig generiert. Diese Züge müssen sortiert werden und Züge, die schon probiert wurden müssen ausmaskiert werden. Der Gewinnerzug wird in einem sechsstufigen Komperatorbaum bestimmt. Sortierkriterien sind dabei die Werte der angegriffenen Figuren oder ob ein Zug ein sogenannter Killerzug ist. Unser Zuggenerator ist somit eine Erweiterung des Deep Blue Zuggenerators [14].

4.2.2.6 Bewertungsfunktion in Hardware:

Die Bewertungsfunktion von Hydra besteht aus vielen kleinen Merkmalen, die unabhängig voneinander berechnet und dann in einem großen Addierbaum zusammengefügt werden. Eine unserer Beobachtungen ist, dass Bewertungsmerkmale, die in Schachbüchern veröffentlicht sind, sowie Allerwelt-Weisheiten für die Konstruktion einer guten Bewertungsprozedur eher hinderlich als hilfreich sind. Viele bekannte Merkmale haben nur für solche Stellungen Aussagekraft, die in Partien wirklich auftauchen. Wenn man 100 Millionen Stellungen pro Sekunde betrachtet, untersucht man aber viele Stellungen, die ‚noch nie ein Mensch zuvor gesehen hat'. Es ist in diesem Fall besonders wichtig, nur universelle Bewertungs-Merkmale zu verwenden. Hydra's Bewertung setzt sich daher im Wesentlichen aus den folgenden Merkmalen zusammen:

- *Spielphase*: In den unterschiedlichen Spielphasen wie Eröffnung, Mittelspiel oder Endspiel haben verschiedene Merkmale unterschiedliche Gewichtungen
- *Material, Materialrelationen*: Die Figurenwerte werden zusammengezählt und für die Kombination von Turm und Läufer sowie für die Kombination Dame und Springer gibt es Boni. In Turm und Läuferendspielen werden die Materialwerte von Bauern angepasst.
- *Angriffsfelder*: Mit Hilfe eines GenVictim-ähnlichen Modules wird erzeugt, welche Figuren angegriffen werden.
- *Zentrumskontrolle*: Derjenige, der das Zentrum kontrolliert, bekommt einen Bonus.
- *Freibauern*: Freibauern bekommen Boni, abhängig von ihrer relativen Position zu den Königen sowie abhängig davon, ob sie die Unterstützung anderer Bauern haben. Auch ob ein Bauer im Quadrat des gegnerischen Königs steht, wird ermittelt.
- *Bauern*: Die Bauernbewertung hängt davon ab, ob ein Bauer isoliert steht, ob er ein Doppelbauer ist, ob er einen eigenen Bauern neben sich hat, ob er von einem eigenen Bauern gedeckt ist und auf welcher Zeile und Spalte er steht.
- *Mobilität*: Es werden die Felder gezählt, die König, Dame, Läufer und Springer von der aktuellen Stellung aus erreichen können.
- *Raumvorteil*: Es werden die kontrollierten Felder gezählt.
- *Ein Turm gehört hinter Freibauern.*
- *Königsangriff*: Es werden die um einen König angegriffenen Felder gezählt.

- *Fesselungen* an Turm und Dame und König werden ermittelt und gezählt.
- *Königs-Schutzschild*: Es wird ermittelt und bewertet, ob die Bauernstellung vor dem König noch in Ordnung ist.

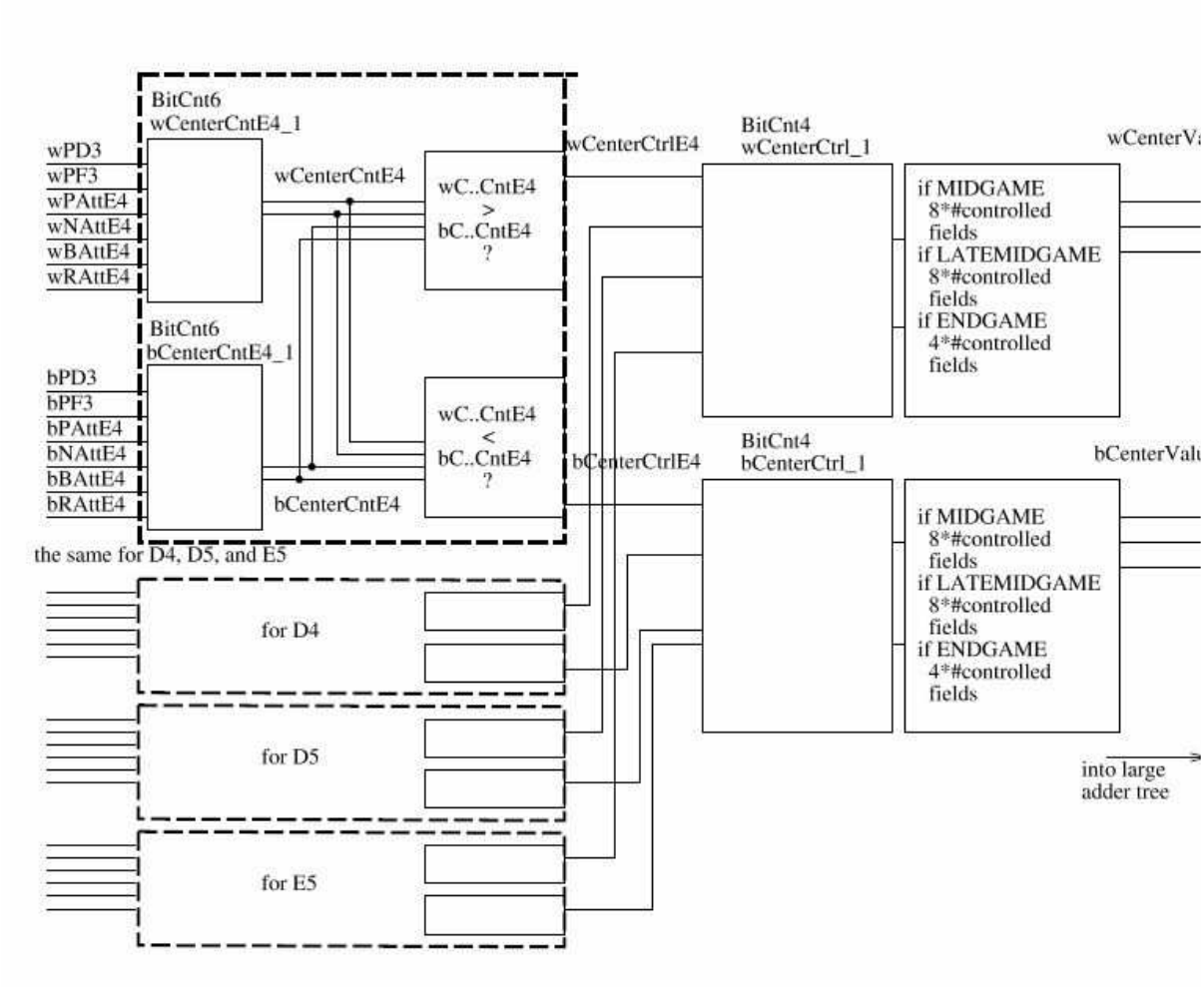


Abbildung 8: Zentrumskontrolle

Die Zentrumskontrolle ist das kleinste Bewertungsmerkmal und eignet sich, um die Vorgehensweise einer in Hardware kodierten Bewertungsfunktion zu illustrieren. Das Modul CenterControl (Abb. 8) bekommt ein paar Inputsignale, die etwas über die Bauernpositionen und über Angriffe auf die Zentrumsfelder aussagen. BitCnt6 ist ein paralleler Addierer von 6 1-bit Variablen und zählt für jedes Zentrumsfeld, wie oft jede Partei dieses Feld angreift. Derjenige, der z.B. das Feld E4 öfter angreift, gewinnt dieses Feld. Derjenige, der mehr Zentrumsfelder für sich entscheiden kann, gewinnt das Zentrum und bekommt dementsprechend Bonuspunkte.

In der linken oberen Ecke von Bild 8 sehen wir innerhalb des gestrichelt gezeichneten Bereichs das Untermodul ‚wCenterCntE4'. Die oberen sechs Inputs von links zeigen an, ob das Feld E4 von weißen Bauern auf D3 oder F3 oder von einer anderen Figur angegriffen wird. Darunter das Modul ‚bCenterCntE4', das die Anzahl der schwarzen Angriffe auf E4 berechnet. Die Outputs sind 3-bit Leitungen, die am rechten Rand der gestrichelten Box miteinander verglichen werden. Als nächstes wird gezählt, wer mehr von den Feldern D4, E4, E5, und D5 für sich entscheiden konnte. Dazu wird ein 4-bit Zähler benutzt. Die ja-oder-nein

Entscheidung darüber, wer das Zentrum gewonnen hat, wird je nach Spielphase noch gewichtet und dann geht der Gesamtoutput in den erwähnten Addierbaum.

4.2.2.7 Experimente

Experimente teilen wir in drei Gruppen. Zunächst einmal gibt es die Speedup-Tests, mit denen wir ermitteln, wie effizient unsere Parallelisierung funktioniert. Dann gibt es Testspielserien mit zwei Prozessoren gegen eine Zweiprozessor-Variante des PC-Schachprogramms Shredder, das unseren Erfahrungen nach als Sparringspartner für Hydra am besten geeignet ist. Darüber hinaus, sozusagen als Ultima-Ratio-Bewertung, messen wir uns im Wettbewerb auf öffentlich einsehbaren Turnieren.

Speedups haben wir mit Hilfe der 30 Schachstellungen aus dem BT2630 Testset ermittelt. Die Messzeiten wurden dabei so angelegt, dass die 32 Prozessor-Version ungefähr zwei Minuten pro Stellung rechnen konnte. Die 32-Prozessor-Version gibt also für jede Teststellung Zwischenziele vor, die auch die anderen Versionen erreichen müssen.

	Zeit(s)	Speedup	SO %
1	53735	1	0
2	32533	1,7	4,5
4	16281	3,3	4,5
8	8777	6,1	8,4
16	4712	11,4	13,0
32	3013	17,8	18,7

Wir vergleichen die Laufzeiten und die Anzahl benötigter Suchknoten der parallelen n-Prozessor-Version von Hydra mit denen der Einprozessor-Version. Der Speedup (SPE) ist die Summe der Zeiten der sequentiellen Version geteilt durch die Summe der Zeiten der parallelen Version. Der Suchoverhead (geleistete Arbeit in % gibt an, wie viele Suchknoten die parallele Version zu viel besucht hat) wird in Prozent angegeben, ausgehend von 100% der sequentiellen Version.

Interne Testpartien zeigen, dass Hydra ca. 200 Elo-Punkte[1] stärker spielt als unsere Standard-Testgegener Shredder9 und Fritz8, die jeweils auf einem Dual-Opteron mit 2,3 GHz rechnen. Die eigentliche und absolut unanfechtbare Leistungsbewertung erhalten wir jedoch durch die Teilnahmen an Schachturnieren.

Hydra's Vorgänger Brutus gewann das Lippstädter FIDE-Großmeisterturnier mit 9 von 11 möglichen Punkten und erreichte 2003 dabei 2768 Elo.

Hydra/Chimera nutzte 16 Prozessoren und 16 langsamere VirtexI FPGAs. Es konnte in Abu Dhabi 2004 gegen das Programm Shredder mit 5,5 zu 2,5 gewinnen. Es schlug den Großmeister Vladimirov (2650 Elo) mit 3,5 zu 0,5. In Bilbao, 2004, holte es ebenfalls 3,5 aus 4 Punkten, gegen einen gegnerischen Elo-Schnitt von 2690 Punkten. Außerdem gewann Hydra das 14. Internationale Paderborner Computerschachturnier mit 8 aus 9.

[1] Das Elo-System ist ein statistisches Maß für die Spielstärkemessung von Schachspielern. Anfänger haben ca. 1000 Elo, Int. Meister 2400, Großmeister 2550, der menschliche Weltmeister ca. 2820 Punkte. Eine 100 Punkte Differenz entspricht einer Gewinnwahrscheinlichkeit von ca. 64%.

Hydra/Scylla, die neueste Version, hatte seinen ersten Auftritt in London 2005. Dort schlug das Programm den englischen Großmeister Michael Adams (2740 Elo) mit 5,5 zu 0,5 und erzielte damit eine Turnierleistung von 3140 Elo.

4.2.2.8 Schlussfolgerung

Wir haben das professionelle Schachprogramm Hydra vorgestellt, das mit sicherlich mehr als 3000 Elo zur Zeit die beste Schacheinheit der Welt ist. Es nutzt FPGA-Technologie und kombiniert Hardware-Design-Methoden von Deep Blue mit Ansätzen völlig dynamischer Baumsuche. Die FPGA-Technologie scheint ein guter Kompromiss zwischen Geschwindigkeit und Flexibilität zu sein. Auf jeden Fall ist es möglich, feingranulare Parallelität in der Anwendung zu heben, und darüber hinaus bleiben einem langwierige Entwicklungszeiten von ASICs erspart. In stark umkämpften Märkten wie dem Computerschach ist es essentiell wichtig, die Möglichkeit zu besitzen, sich schnell ergebende Änderungen in den Programmcode zu implementieren

4.3 Planung unter Unsicherheit

4.3.1 Literatur zu Planung und Planung unter Unsicherheit

Ein Plan ist eine Lösung eines Zuweisungsproblems, das sich über mehrere Zeitschritte erstreckt. Der größte Teil der Forschung im Bereich der Optimierungen von Plänen beschäftigt sich mit der Generierung statischer, vorberechneter Pläne in dem Sinne, dass man annimmt, die zum Planungszeitpunkt vorliegenden Plandaten seien vollständig bekannt und unveränderbar. Traditionellerweise werden Pläne erzeugt, die das Ziel haben, den Gewinn unter Verwendung von Schätzdaten bzw. „Erwartungsdaten“ zu maximieren.
Sobald ein vorberechneter Plan in der echten Welt eingesetzt wird, werden verschiedene Unsicherheitsaspekte des Systems, in dem der Plan abläuft, zu Störungen führen, so dass der ursprüngliche Plan seine Güte oder sogar seine Zulässigkeit verliert. Es ist dann die Aufgabe des Störungsmanagements, einen teilweise neuen Plan zu entwickeln, der das System wieder in einen zulässigen Zustand überführt. Der alte Plan wird also schnell außer Kraft gesetzt, bzw. muss „repariert“ werden, oder es muss sogar komplett neu geplant werden.

Da die Welt, in der wir leben, immer dynamischer zu werden scheint, d.h. dass Erfahrungswerte von gestern immer weniger geeignet sind als Planungsdaten für morgen herangezogen zu werden, glaube ich, dass es ebenfalls immer wichtiger wird, Datenunsicherheiten mit in die Planung einzubeziehen. Der Grund für Störungen liegt nämlich offensichtlich in der Tatsache begründet, dass Planer Informationen über das reale Verhalten über eine sich verändernde Umgebung entweder nicht haben oder diese bewusst missachten. Oft kennt man Eingabedaten nur ungefähr, man kennt evtl. Verteilungen über die Daten. Bei der Flugplanung kennen wir z.B. eine Verteilung über die möglichen Ankunftszeitpunkte von Flügen, wobei „kennen“ natürlich auch hier lediglich unsere Modellvorstellung widerspiegelt.

Ich möchte Planungsaufgaben deshalb unter dem Aspekt der „Mehrstufige Entscheidungen unter Risikoeinfluss“ betrachten. Es ist ein Teilgebiet des größeren Feldes von entscheidungstheoretischen Ansätzen [13], zu dem auch die (lineare) stochastische Programmierung gehört [8][33]. Manchmal wird der Begriff „robuster Plan“ verwendet, der in zweierlei Bedeutung auftritt: Zum einen informal als Plan, dessen Wert relativ unsensibel

auf mögliche Realisierungen der echten Welt reagiert, aber dann auch als Plan, der auch bei schlimmstmöglichem Verlauf der exogenen Einflüsse noch gültig bleibt. Bei robuster Planung (vgl. [41]) geht man also von grundsätzlicher Risikoscheu des Anwenders bei Entscheidungssituationen mit ausgeprägter Unsicherheit der verfügbaren Informationen aus. Scholl [41] unterscheidet zudem zwischen robuster Planung, bei der die Robustheit als zentrales Bewertungskriterium im Mittelpunkt steht, und robuster Optimierung, die Modelle und Methoden zur Verfügung stellt, um robuste Pläne zu erzeugen.

4.3.2 Spielbaumsuche zur Planung unter Unsicherheit

Wir sehen unsere Arbeiten im Bereich zwischen robuster Optimierung und stochastischer Optimierung, wobei wir uns zunächst am Erwartungswert orientieren und nicht an Minmax-, (μ,σ)-, Bernoulli- oder anderen Kriterien [6][21][28][41].

Um sinnvoll über „Planung unter Unsicherheit“ mit Hilfe von Spielbäumen diskutieren zu können, müssen wir klären, was „Pläne“ in einem Spielbaum sein sollen. Ausgangspunkt ist ein ganzzahliges mehrstufiges (lineares) Entscheidungsproblem mit seinem Spielbaum, den man auch „vollständigen stochastischen Entscheidungsbaum“ nennt. Im Gegensatz zur „stochastischen Programmierung“ setze ich an, bevor das gegebene Problem in die Form eines linearen Ungleichungssystems gebracht wird.

Sei also ein Baum **T** gegeben, der alle möglichen Zustände sowie unsere möglichen Aktionen als auch die möglichen Aktionen der Natur im Vorausschau-Zeittrichter wiedergibt. Er bestehe aus zwei verschiedenen Knotentypen, Max-Knoten (rechteckig) und Avg-Knoten (dreieckig). Herausgehende Kanten aus Max-Knoten repräsentieren unsere möglichen Aktionen, herausgehende Kanten aus Avg-Knoten repräsentieren die Fähigkeit der Natur, auf verschiedene Arten zu agieren. Jeder Pfad von der Wurzel zu einem Blatt kann als eine mögliche traditionelle Lösung unserer Planungsaufgabe angesehen werden. Unsere Aktionen sind definiert durch die Kanten, die wir an Max-Knoten nehmen, immer unter der Annahme, dass die Natur eine eindeutige, vorher bekannte Kante aus den AVG-Knoten auswählt.

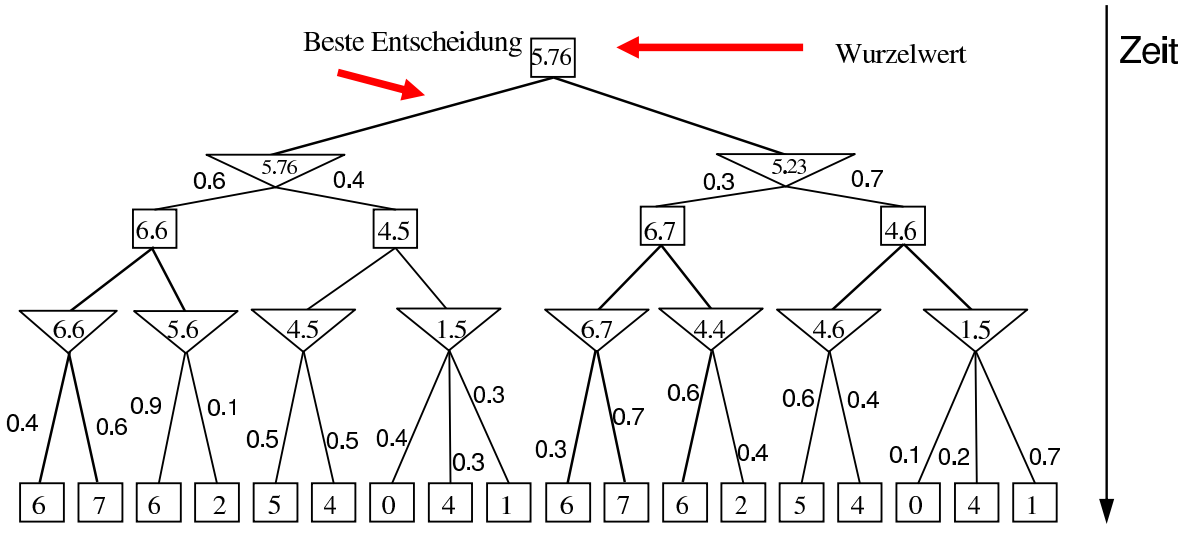


Abbildung 9: Spielbaum, Maximierer gegen Natur

Wir nehmen nun an, dass Kostenwerte an den Blättern des Baums bekannt sind. Sie

repräsentieren die totalen Kosten des „Planungspfades“ von der Wurzel zum Blatt. Der Wert an inneren Max-Knoten errechnet sich als das Maximum der Werte der Nachfolgeknoten. Der Wert an inneren Avg-Knoten wird gebildet, indem ein gewichteter Durchschnittswert über die Werte der Nachfolgeknoten gebildet wird (vgl. Abb. 9).

Sei eine so genannte *Max-Strategie* **S** (Abb. 10), auch Kontrollstrategie genannt, wieder ein Teilbaum von **T**, der die Wurzel von **T** sowie genau einen Nachfolger an Max-Knoten und alle Nachfolger an Avg-Knoten enthält. Jede Strategie **S** habe einen Wert f(**S**), der als der Wurzelwert von **S** bzgl. des (Teil-)Baumes **S** definiert ist. Eine so genannte Hauptvariante

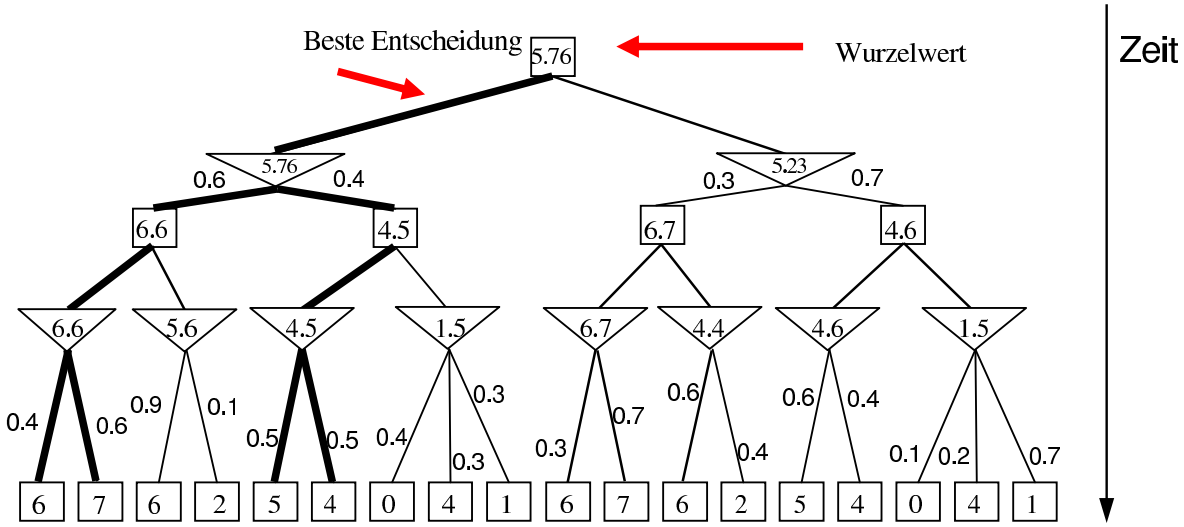


Abbildung 10: Strategie im Spielbaum, Maximierer gegen Natur

p(**S**), auch *Interner Plan* genannt, einer solchen Max-Strategie kann dadurch bestimmt werden, dass wir die Kanten, die die Max-Knoten verlassen, sowie die am höchsten gewichtete Kante jedes Avg-Knotens auswählen (Abb. 11). Der Pfad, der sich von der Wurzel ausgehend zu einem Blatt ergibt, ist p(**S**). Wir sind an dem internen Plan p($\mathbf{S_B}$) der

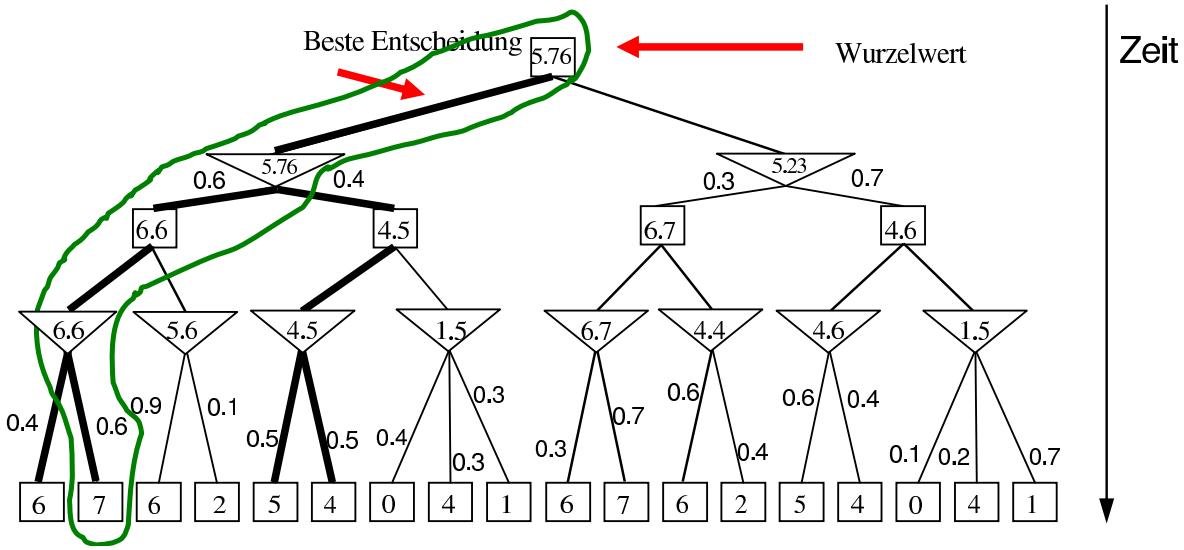


Abbildung 11: Plan im Spielbaum, Maximierer gegen Natur

besten Strategie $\mathbf{S_B}$ sowie den erwarteten Kosten E($\mathbf{S_B}$) von $\mathbf{S_B}$ interessiert. Trivialerweise sind die Kosten E($\mathbf{S_B}$) gleich dem Wurzelwert von **T**. Die erwarteten Kosten E(**p**) eines internen Plans **p** werden definiert als die erwarteten Kosten der besten Strategie **S**, zu der der interne Plan **p** gehört. Es ist also E(**p**) = min{E(**S**) | p(**S**) = **p**}. Wir nennen einen internen Plan **p** *optimal*, wenn gilt **p** ∈ {**p'** | E(**p'**) ist maximal}. Oft wird auch nur der beste Zug, eine Kante von der Wurzel zu einem Nachfolger mit gleichem Wert, wie die Wurzel hat, benötigt.

Anmerkung:
Wie bereits im SSAT-Beispiel gesehen, bleibt Entscheidungsfindung unter Unsicherheit im Allgemeinen PSAPCE-hart, selbst wenn wir eine interessante, in der stochastischen Optimierung übliche Vereinfachung vornehmen, nämlich, dass der Avg-Spieler unabhängig von den Aktionen des Max-Spielers agiert. Das hieße im obigen Beispiel unter anderem, dass die Natur, wenn sie im linken Ast unter der Wurzel mit Wahrscheinlichkeit 0,6 nach links und mit Wahrscheinlichkeit 0,4 nach rechts zieht, im rechten Ast unter der Wurzel nicht mit Wahrscheinlichkeit 0,3 nach links gehen kann. Sie müsste auch hier mit Wahrscheinlichkeit 0,6 nach links gehen.

4.3.3 Das Reparaturspiel

Es sind drei Beobachtungen, die uns der Anwendung von Spielbaumsuche zur Planung unter Einbeziehung von Datenunsicherheit entscheidend näher bringen.

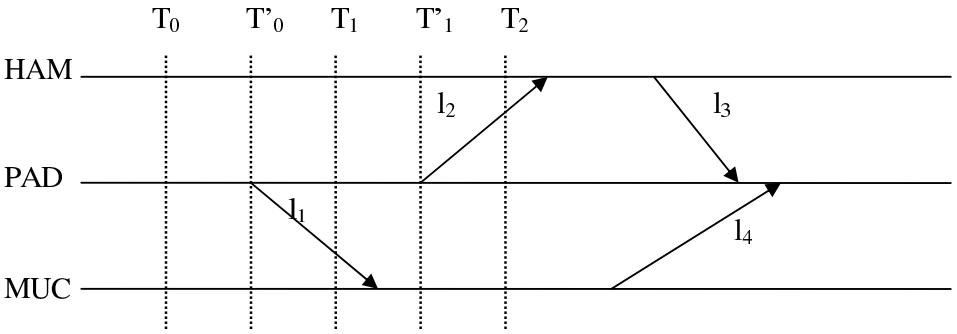
Beobachtung 1: Der interne Plan ist so definiert worden, dass er sich als ein Pfad des Spielbaums ergibt. Das beinhaltet, dass zu jedem Zeitpunkt, an dem der Max-Spieler am Zug ist, nur ein Teil des Plans festgelegt wird. Häufig wird aber zu jedem dieser Zeitpunkte eine vollständige Zuweisung aller Resourcen (z.B. Flugzeuge) auf Aufgaben (z.B. Flugsegmente) für den gesamten Zeithorizont, also ein vorberechneter Plan im traditionellen, deterministischen Sinne, benötigt. Man braucht so einen Plan, um langfristige Ziele optimieren zu können, bzw. um sich mit dessen Hilfe mit der Umwelt zu arrangieren. Z.B. wird in der Flugplanung diese Art Plan benötigt, damit die Kunden wissen, wann sie an welchem Ort zu sein haben. Ein einzelner Zug des Max-Spielers kann somit auch die Auslegung eines Plans sein.
Um die beiden Planbegriffe verbal unterscheiden zu können, nennen wir Pläne, die Züge im Spiel darstellen, **Ausgabepläne**.

Beobachtung 2: Weil Unterschiede zwischen Abläufen gemäß dem Ausgabeplan und tatsächlichen Abläufen Kosten verursachen, sind die zu erwartenden Kosten (bzw. Gewinne), die zu einem Ausgabeplan gehören, nicht dieselben, bevor und nachdem ein Ausgabeplan an Kunden und Zulieferer herausgegeben wurde. Ein Plan erhält einen eigenen Wert, nachdem er veröffentlicht ist und andere Parteien von ihm abhängen. Ich halte dies für eine zentrale Beobachtung, die für viele Planungsaufgaben unter Unsicherheit gültig ist. Einen Plan macht man nicht aus Freude an Ablauffestlegungen, sondern ein Plan macht nur dann Sinn, wenn er eine gewisse Verlässlichkeit besitzt. Änderungen am Plan mögen unerlässlich sein, aber sie sind teuer und das sollte zum Planungszeitpunkt mit berücksichtigt werden.

Beobachtung 3: Im Prinzip kann sich ein Ausgabeplan beliebig von einem internen Plan unterscheiden. Ein Ausgabeplan braucht noch nicht einmal ein gültiger interner Plan zu sein, da er ja lediglich Kommunikationszwecken dient.
Wenn bekannt ist, dass die Eingabedaten zwar nicht deterministisch sind, aber doch so, dass Erwartungswertdaten bei der Planung gute Hilfsgrößen sind, verlangen wir von Ausgabeplänen, dass sie selber einen gültigen internen Plan darstellen.

D.h.: Das **Planungsproblem**, mit dem wir es im Folgenden zu tun haben, sieht so aus, dass an der Wurzel des Baums zunächst ein Ausgabeplan zu wählen ist, für den es einen gewissen Gewinn zu verbuchen gibt; in den folgenden Stufen reduziert sich aber dieser Gewinn aufgrund von Umplanungen. Eine Umplanung bedeutet zum einen, dass ein neuer Rest-Ausgabeplan festgelegt wird, und zum anderen, dass der interne Plan des Unternehmens auf der besten Ursprungsstrategie umgebogen wird. Diese Sonderform der Planung unter Unsicherheit nennen wir das **Reparaturspiel**, wenn zusätzlich gilt, dass Ausgabepläne auch gültige interne Pläne sind.

Ein kleines Flugplanbeispiel (Abb. 12) soll das erläutern:

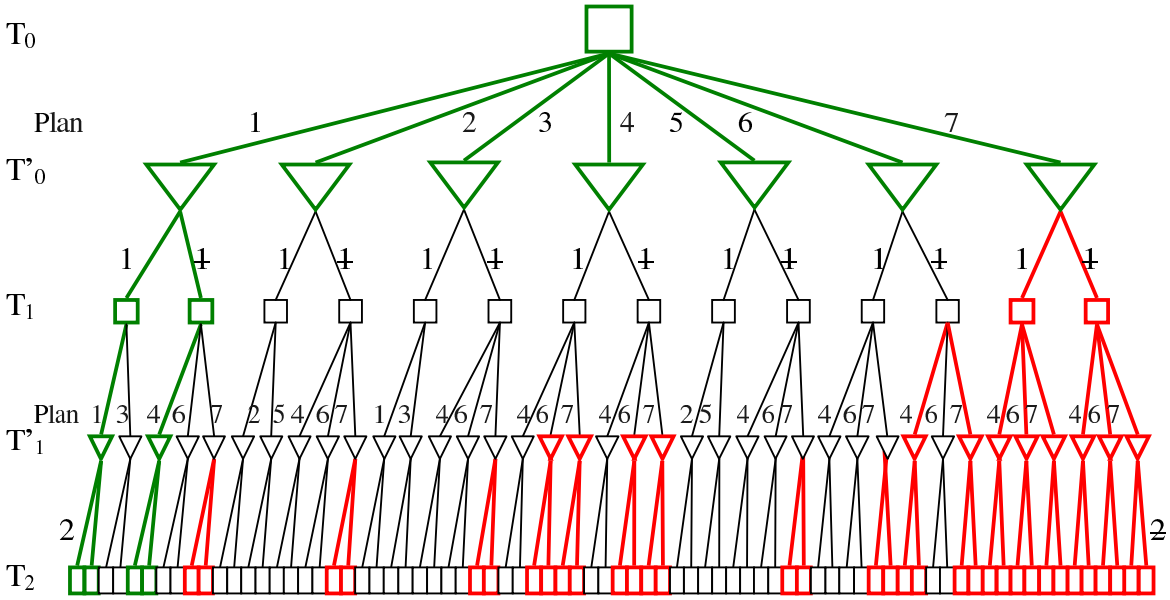


Seien l_1, l_2, l_3 und l_4 Flugsegmente und seien F_1 und F_2 Flugzeuge. Wir können zum Planungszeitpunkt nicht sicher sein, dass l_1 und l_2 tatsächlich geflogen werden können. Zum Zeitpunkt T_0 müssen wir einen Ausgabeplan festlegen, der alle Flugsegmente umfasst. Bei T'_0 erfahren wir, ob l_1 geflogen werden kann oder ob der Flug gestrichen wird. Ein neuer Ausgabeplan muss bis zum Zeitpunkt T_1 erstellt sein, und bei T'_1 erfahren wir, ob l_2 stattfindet. Bei T_2 geben wir den endgültigen Flugplan heraus.

Ausgabeplan 1: Flugzeug F_1 fliegt l_1 und l_4, F_2 fliegt l_2 und l_3.
Ausgabeplan 2: Flugzeug F_2 fliegt l_1 und l_4, F_1 fliegt l_2 und l_3.
Ausgabeplan 3: F1 fliegt l_1 und l_4, F_2 bleibt in Paderborn (PAD)
Ausgabeplan 4: F1 fliegt l_2 und l_3, F_2 bleibt in Paderborn (PAD)
Ausgabeplan 5: F2 fliegt l_1 und l_4, F_1 bleibt in Paderborn (PAD)
Ausgabeplan 6: F2 fliegt l_2 und l_3, F_1 bleibt in Paderborn (PAD)
Ausgabeplan 7: kein Flug wird geplant

Abbildung 12: Flugplanungsbeispiel

Abb. 13 verdeutlicht, zu welchen Zeitpunkten wir welche Ausgabepläne zur Verfügung haben und welche Sequenzen von Ausgabeplänen entstehen können.



1 bedeutet: Flugsegment 1 darf stattfinden, ~~1~~ bedeutet, Flugsegment 1 darf nicht stattfinden. 2 und ~~2~~ analog.

Abbildung 13: Spielbaum des Flugplanungsbeispiels

Natürlich werden wir bei der Frage, wie wir das gegebene Spiel auswerten, nicht den vollständigen Spielbaum auswerten wollen, der nur implizit gegeben ist. Man kann sich schnell überlegen, dass man in der Tat auch nicht alle Knoten betrachten muss. Im günstigsten Fall braucht man nur die grünen Knoten zu betrachten. Dabei bestimmt der grün gefärbte Teilbaum unterhalb von Plan1 an der Wurzel eine untere Schranke für den Wert von Plan 1 an der Wurzel. Wenn die übrigen Wurzelnachfolger schon in Ebene 1 nicht mehr Gewinn versprechen als die untere Schranke von Plan 1, ist klar, dass der ganze Rest vernachlässigt werden kann.
Die roten Knoten brauchen auch im schlimmsten Fall nicht untersucht zu werden. Bei der ersten Berührung mit einem roten Knoten, kann man feststellen, dass der darunterhängende Teilbaum unwichtig für das Gesamtergebnis ist. Die roten Knoten können weggeschnitten werden, weil die darüber gefällten Entscheidungen den Szenarienraum derart partitionieren, dass eine Untersuchung dieser Teilbäume überflüssig ist. Alle anderen Knoten müssen je nach der Verteilung von Werten auf die Blätter bewertet werden. Ein Alphabeta-artiger Ansatz kann hier weitere Beschneidungen des Suchbaums bringen.

Die Auswertung des Spielbaums macht Informationen über mögliche zukünftige Ereignisse nutzbar, um die Ausführung von Aktionen zu kontrollieren. So eine Kontrollstrategie ist von großer praktischer Bedeutung, da sie es ermöglicht, statische Pläne in unsicheren Umgebungen zu nutzen. In diesem Sinne ist eine Kontrollstrategie in dieser Arbeit ähnlich zu einer Kontrollstrategie in [22]. Die dort vorgestellte Umsetzung der Kontrollstrategie auf Job-Shop Scheduling ist allerdings nicht direkt auf unsere Problemstellung übertragbar, da zum einen keine Unterscheidung von internen Plänen und Ausgabeplänen gemacht werden, und sich damit zum anderen nicht das Problem der Kostenmessung von Umplanungen ergibt. Man unterschied dort nur zwischen Kosten und Gewinnen, die vom Systemzustand abhängen, aber nicht Kosten und Gewinne, die von Wegen in ein und denselben Zusand abhängen.

Definition Reparaturspiel

Wir definieren das Reparaturspiel über seinen Spielbaum $G=(V=V_{avg} \cup V_{max},E,h: V \rightarrow \mathbb{N}_0)$ mit den Blättern $L(T) \subset V$. Seine Auswertung gibt uns ein Maß für den Wert eines Ausgabeplans bei unsicherer Datenlage.
Der Wert des Spiels ist definiert als der so genannte weighted max-avg-Wert wmaxav:

wmaxav-Wert. Sei G=(V, E, h) ein Spielbaum und w_v:N(v) → [0,1] sei eine Gewichtsfunktion für alle $v \in V_{AVG}$, N(v) die Menge aller Nachfolger des Knotens v. Die Funktion wmaxav: V → $\mathbb{N}_0$ ist rekursiv wie folgt definiert.

$$wmaxav := \begin{cases} h(v) & , falls\ v \in L(G) \\ max\{ wmaxav(v') \mid v' \in N(v)\} & , falls\ v \notin L(G)\ und\ v\ MAX\text{-}Knoten \\ \sum_{v' \in N(v)} (w_v(v', wmaxav(v')) \cdot wmaxav(v')) & , falls\ v \notin L(G)\ und\ v\ AVG\text{-}Knoten \end{cases}$$

Dabei soll gelten, dass $\sum_{v' \in N(v)} w_v(v') = 1$ ist, allerdings darf w_v eine Funktion sein, bei der das Gewicht auch von der Güte von wmaxav(v') abhängen kann.

Reparaturspiel. Der Wert des Reparaturspiels (G,p,g,f,s) ist wmaxav(r) des implizit gegebenen Spielbaums G=(V,E,g+f) mit Wurzel r und uniformer Tiefe t. $p \in L(G)$ ist ein spezielles Blatt und g, f und s sind Funktionen. Kanten aus E können als Ausgabepläne interpretiert werden. Der Spielbaum eines Reparaturspiels hat folgende Eigenschaften:

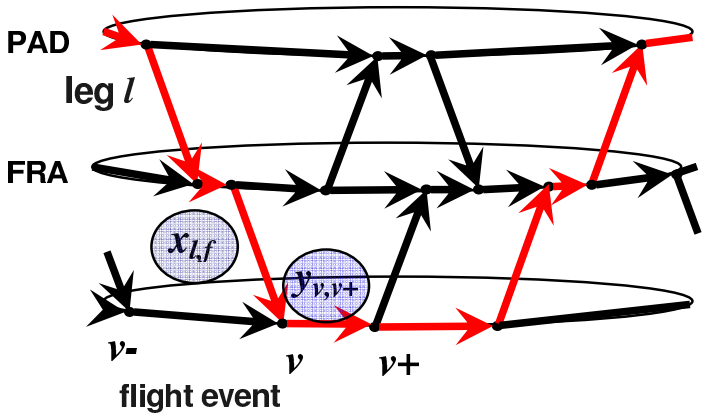
- Sei $P = (r=v_1, v_2,\ldots, p=v_t) \in V^t$ der eindeutige Pfad von r nach p. P beschreibt den initialen internen Plan der Firma. Wir nennen ihn auch den *Masterplan*.
- V ist partitioniert in Mengen $S_1,\ldots, S_n$, $|V| \geq n \geq t$ durch die Funktion s:V → $\{S_i\}_{1 \leq i \leq n}$. Alle Knoten, die zur selben Partition S_i gehören, sind im selben *Systemzustand* – z.B. in der Flugplanung: welches Flugzeug ist wann wo – unterscheiden sich aber in ihren Historien, die in diesen Zustand geführt haben und damit von der Folge von Ausgabeplänen, die bislang veröffentlicht wurden.
- Die Funktion g: $\{S_i\}_{1 \leq i \leq n}$ → $\mathbb{Z}$ spezifiziert die erwarteten zukünftigen Kosten für Knoten, abhängig nur von ihrem Zustand.
- f: $\bigcup_{1 \leq \tau \leq t} \{V\}^\tau$ → $\mathbb{N}_0$ beschreibt die induzierten Reparaturkosten für jeden möglichen Teilpfad im Baum (V,E). Insbesondere hat jeder Teilpfad P' von P Reparaturkosten Null, f(P')=0.
- h: V → N_0 ist definiert als h(v) = g(s(v)) + f(r … v) und stellt die eigentliche Knotenbewertungsfunktion dar.

Die Aufgabe ist es, robuste Pläne zu entwickeln, entweder für das Ausgangsproblem oder für online-Umplanungen, die nach Störungen erzeugt werden. Robuste Umplanungen stellen in gewissem Sinne ein auf der Durchführungsebene stattfindendes reagierendes und zugleich präventives Störungsmanagement vor. Das bevorzugte Verfahren, das wir anwenden wollen, ist eine Baumsuche mit simulativem Charakter, die Spielbaumknoten systematisch untersucht.

4.3.4 Große Flugplanungsinstanz als Beispielanwendung [K14][K13][K8][V1]

4.3.4.1 Deterministische Planung

Verkehrsplanung für Flug- und Bahnverkehr [44][45] sind besonders wichtige Anwendungen im Bereich der industriellen Planung. Zum einen geht es bei der Optimierung von Geschäftsabläufen z.B. bei großen Fluggesellschaften um sehr viel Geld, zum anderen ist in Folge dessen und wegen der sehr starken Konkurrenzsituation bereits sehr viel Forschung in diesem Bereich betrieben worden, was wiederum dazu führt, dass wir ein sehr hohes Vertrauen in die Qualität der dort eingesetzten Methoden haben. In der Flugplanung dominieren mathematische Modelle und Methoden zu deterministischer Planung, welche eine großartige Erfolgsgeschichte für alle Arten von logistischen Planungsproblemen feiern. Das ganze Spektrum des Operations Research wird genutzt, um anstehende Planungsprobleme insbesondere mit sehr großen LP und IP Modellen zu lösen.
Der Planungsprozess von Fluggesellschaften [4] beginnt mit dem so genannten Netzwerk-Design, bei dem grob festgelegt wird, welche Flugverbindungen mit welcher Häufigkeit bedient werden sollen. Daraus wird ein erster „Plan“, das Flug-Schedule, generiert, der festlegt, welche Non-Stop-Flugverbindungen (*Flüge, Legs*) zu welchen Zeiten den Passagieren angeboten werden sollen. Danach folgen diejenigen Planungsprozesse, die wir im Folgenden



$$\text{maximize} \quad \sum_{l\in\mathcal{L}} \sum_{f\in\mathcal{F}} p_{l,f} \cdot x_{l,f}$$

subject to

$$\begin{aligned}
\sum_{f\in\mathcal{F}} x_{l,f} &= 1 && \forall\, l \in \mathcal{L} \\
\sum_{l_f^{arr}=v} x_{l,f} - \sum_{l_f^{dep}=v} x_{l,f} + y_{v^-,v} - y_{v,v^+} &= 0 && \forall\, v \in V \\
\sum_{l_f\in E_{F0}^f} x_{l,f} + \sum_{(v,v^+)\in E_{G0}^f} y_{v,v^+} &\le size_f && \forall f \in \mathcal{F} \\
x_{l,f} &\in \{0,1\} && \forall\, l \in \mathcal{L}; \forall\, f \in \mathcal{F} \\
y_{v,v^+} &\ge 0 && \forall\, (v,v^+) \in E_G
\end{aligned}$$

Abbildung 14: Deterministisches Fleetassignment

genauer betrachten wollen. Im Allgemeinen besitzen Fluggesellschaften Flugzeuge unterschiedlichen Typs (*Flotten*). Sie unterscheiden sich in ihrer Sitzkapazität und anderen wirtschaftlich relevanten Kenngrößen. Die Aufgabe des Fleet Assignments ist es, aus einem gegebenen Flug-Schedule und einer Menge von Flotten eine gewinnmaximale Zuweisung von Flugzeugtypen zu den einzelnen Flügen zu bestimmen. Damit beantwortet eine Lösung des Fleet Assignment Problems (FAP) die Frage, wie viele Flugzeuge von welchem Typ sich wo zu welcher Zeit befinden sollen. Vom FAP ist bekannt, dass es in einigen Varianten NP-vollständig und nicht approximierbar ist [11].
So genannte Time-Space Netzwerke (vgl. Abb. 14), bei denen es sich um spezielle Fluss-Graphen handelt, können verwendet werden, um eine mathematische Formulierung für das FAP zu erzeugen. Dieser Ansatz wurde in [12] zum Lösen des Fleet Assignment Problems erstmals eingesetzt und ist seitdem von verschiedenen Autoren modifiziert und erweitert worden [10][17]. Das sich ergebene Min-Cost-Flow Problem auf dem Time-Space Netzwerk wird typischerweise in ein gemischt-ganzzahliges lineares Programm (MIP) transformiert und mit Standardsoftware (z.B. ILOG CPLEX) gelöst.

In einem nächsten Schritt wird aus der FAP-Lösung ein so genannter Rotationsplan erzeugt. Ein solcher Rotationsplan legt für jedes (physikalische) Flugzeug genau fest, zu welcher Zeit es sich an welchem Ort der Welt aufhalten soll. Anschließend folgen weitere Planungsschritte, mit denen wir uns hier nicht weiter beschäftigen wollen: Die Flugzeugbesatzungen werden zum Beispiel im so genannten Crew-Rostering und Crew-Pairing den Flugzeugen zugewiesen.
Fleet Assignment und Rotationsplanung gehören zu den lang- bis mittelfristigen Planungsphasen einer Fluggesellschaft (1 Jahr bis 2 Monate vor Operations). Sie liefern (Flugzeug-)Pläne unter Berücksichtigung von ökonomischen Parametern (Passagieraufkommen, Kosten, Erlöse, …), fluggesellschaftsspezifischen Parametern (Anzahl verfügbarer Flugzeuge, Sitzkapazitäten, Verfügbarkeit von Flugbesatzungen, …) und operationellen Einschränkungen (Flugdauern, Wartungsvorschriften, …). Das Ziel ist es dabei, den Gesamtgewinn zu maximieren. Um die Wirtschaftlichkeit zu erhöhen, wird unter Einsatz moderner Optimierungsverfahren die Auslastung der Flugzeuge mehr und mehr erhöht, was zur Folge hat, dass die Flugzeugumläufe immer weniger Pufferzeiten aufweisen. Je kleiner die Pufferzeiten in den Umläufen werden, desto häufiger treten allerdings Störungen auf, die vom Operations-Control-Management behoben werden müssen [34][35]. Im Falle von Fluggesellschaften führen sie zu Verspätungen, geänderten Flugzeugumläufen, Flugstreichungen, Slot-Problemen usw. Zusammengefasst: Störungen verursachen Schwierigkeiten und Kosten. Deshalb sollte es ein wichtiges Kriterium der lang- und mittelfristig arbeitenden Planungsabteilungen sein, Pläne zu erzeugen, die im Falle von Störungen schnell und Kosten sparend repariert werden können.

4.3.4.2 Aufgabenstellung und Erfolgskontrolle

Ausgangspunkt unserer Untersuchungen ist das von uns so genannte *Stochastische Fleetassignment-Problem*. Es besteht aus der Beschreibung des deterministischen Fleetassignment-Problems, plus für jedes Leg zwei diskrete Wahrscheinlichkeitsverteilungen, die besagen, mit welchen Wahrscheinlichkeiten die Dauer eines Flugsegments um wie viele Minuten verlängert wird, bzw. ob es ausfallen muss. Außerdem wird die Zeit in d-minütige Intervalle diskretisiert.

Auf der einen Seite können wir, wenn wir mit Modellen und Methoden der Mathematik und Informatik in der realen Welt Erfolg haben möchten, nicht davon ausgehen, dass es ausreicht, Modelle und Problemstellungen aufzustellen, die in Polynomzeit zu lösen sind. Wenn wir

andererseits unsere Aufmerksamkeit auch auf „schwere“ Probleme ausweiten wollen, brauchen wir geeignete Erfolgskontrollen. Im Falle des oben gegebenen stochastischen Flottenzuweisungsproblems, haben wir uns mit unserem industriellen Partner auf die Evaluation in einer Simulationsumgebung geeinigt, die wie folgt aussieht:

Zunächst haben wir die zu simulierende Zeit in Intervalle von d = 15 Minuten diskretisiert. Die Basis für unsere Simulation und die Eingabe für den Simulator stellt ein kontinentaler Lufthansa Flugplan dar. Zusätzlich stehen uns weitere Daten zur Verfügung, um alternative Pläne für die Reparatur berechnen zu können. Der Plan besteht aus 20603 Flugsegmenten, die von 144 Flugzeugen aus 6 Flotten bedient werden. Der Simulator bewegt nun in d-minütigen Zeitsprüngen eine virtuelle Zeitlinie über den gegebenen Plan und erstellt Störungen. Jeder Abflug des aktuellen Umlaufplans stellt ein mögliches Ereignis dar, und alle Ereignisse, die innerhalb eines d-Minuten-Intervalls liegen, werden als gleichzeitig stattfindend angesehen. Wenn während der Simulation ein Ereignis (ein abgehender Flug) auftritt, kann der betroffene Flug gestört werden, d.h. er kann um 30, 60 oder 120 Minuten verzögert werden oder auch ganz aus dem Schedule gestrichen werden. Die Störungen treten dabei jeweils mit einer vorgegebenen Wahrscheinlichkeit auf. Bezeichne T den aktuellen Simulationszeitpunkt. Der Simulator betrachtet und stört gegebenenfalls alle Ereignisse im Intervall [T,T+d], übergibt die Störungen an eine Reparaturengine, wartet auf den reparierten Flugzeug-Umlaufplan und geht d Minuten in der Zeit vorwärts.

Unsere Aufgabe besteht nun darin, während der Simulation die auftretenden Störungen möglichst kosteneffizient und Gewinn bringend über den zu simulierenden Zeitraum hinweg zu beheben.

4.3.4.3 Das Fleet-Assignment-Reparaturspiel

Ein wichtiger Punkt bei der Anwendung des Reparaturspiels, verbunden mit der Auswertung seines Spielbaums, ist die Frage nach der Komplexität des auf eine Anwendung bezogenen Reparaturspiels. Für viele stochastische Optimierungsprobleme ist eine Baumsuche nämlich nicht das geeignete Mittel, wie ein stochastisches Set-Cover Problem [38] bzw. einige stochastische Schedulingprobleme [27] zeigen, für die es schnelle Approximationsverfahren mit polynomieller Laufzeit gibt.. Wenn die deterministische Variante eines stochastischen Problems bereits NP-schwer und nicht approximierbar ist, gibt es aber kaum Hoffnung auf schnelle Approximationsverfahren für das stochastische Problem.

In der Tat können wir zeigen, dass das Stochastische Fleet-Assignment-Problem PSPACE-vollständig ist. Außerdem wissen wir, dass das deterministische Fleet-Assignment-Problem NP-vollständig und nicht approximierbar ist.

Die Konsequenz ist, dass wir keinen über eine Planungsperiode von mehreren Tagen oder gar Wochen robusten Plan finden können. Stattdessen betrachten wir zukünftige Szenarien nur lokal um einen gegebenen Zeitpunkt herum. Das impliziert jedoch wiederum, dass wir wie beim Schachspiel heuristische Werte an künstlich entstehenden Blättern benötigen. Hier hilft uns eine weitere Beobachtung aus der Praxis, eine weitere Annahme in unser Modell einfließen zu lassen: Nach einer Störung will man auf jeden Fall so schnell und kostengünstig wie möglich in den Ursprungsplan zurückfinden. Die **Umplanungskosten** für einen möglichen neuen Ausgabeplan bemessen sich also nach der Größe des Umweges, den der neue Ausgabeplan braucht, um ab einem bestimmten Zeitpunkt wieder gleich mit dem alten zu sein. Ein vorgegebener Ausgabeplan, der die Aufgabe des Masterplans übernimmt,

ermöglicht uns die Erzeugung heuristischer Blattwerte: Wir können messen, wie weit ein möglicher nächster Ausgabeplan vom Masterplan abweicht.

Das Planungsteam einer Fluggesellschaft soll das Reparaturspiel mit der Erzeugung eines traditionellen Planes für ihre Aktivitäten beginnen. Der Pfad P repräsentiert diesen Plan, der zugleich der am meisten erwartete Plan im Zeittrichter ist und der interessanterweise einen eigenen Wert bekommt, sobald er erst einmal generiert worden ist. Er ist klein, kann kommuniziert werden, und sobald ein Kunde oder Zulieferer diesen Plan empfangen hat, verursacht jede weitere Änderung des Plans Änderungskosten. Wir nehmen diesen Plan als Masterplan her. Störungen im Ablauf können die Luftfahrtgesellschaft nun daran hindern so vorzugehen, wie es ursprünglich geplant war.
Die möglichen Interaktionen von Natur und Gesellschaft bilden einen Spielbaum mit Avg- und Max-Knoten. Avg-Konten sind Naturknoten, und der Avg-Spieler formalisiert diskrete Wahrscheinlichkeitsverteilungen über seine Handlungsalternativen. Jede Störung zwingt die Fluggesellschaft, ihren Plan anzupassen und einen neuen Plan zu produzieren. Der Max-Spieler repräsentiert das Unternehmen selbst. Sobald eine Störung aufgetreten ist, wählt es einen reparierenden Teilplan aus, dessen negative Reparaturkosten plus seine erwarteten zukünftigen Gewinne maximal sind.
Weil der Wert eines Blattes v auch davon abhängt, wie „weit“ der Pfad (r,…,v) vom Pfad P entfernt ist, ist es nicht möglich, Systemzustände – wo ist welches Flugzeug zu welchem Zeitpunkt – mit Baumknoten gleichzusetzen. Deshalb gibt es die Partition $V = S_1 \cup \ldots \cup S_n$. In S_i sind alle Knoten zusammengefasst, die zu einem bestimmten Systemzustand gehören, aber verschiedene Historien haben. Alle Knoten innerhalb einer Partition S_i beschreiben denselben Zustand und haben dieselben erwarteten zukünftigen Reparaturkosten, die durch die Funktion g geschätzt werden. Die Funktion f bewertet für beliebige Teilpfade, wie weit diese von dem Ebenen-korrespondierenden Teilpfad von P entfernt sind. Innere Knoten des Spielbaums werden mit Hilfe der Knotentypen bewertet.

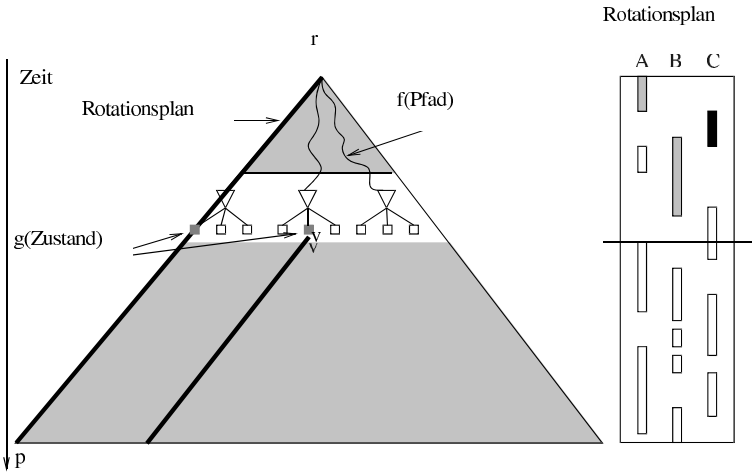


Abbildung 15: Reparaturspiel-Spielbaum und Rotationsplan

Abb. 15 zeigt einen Rotationsplan auf der rechten Seite. Die Flugzeuge A, B und C befinden sich entweder auf dem Boden oder in der Luft, was durch Boxen angezeigt wird. Eine schattierte Box bedeutet, dass der Masterplan verändert wurde. Die Zeit läuft von oben nach unten. Der linke Teil der Abbildung zeigt einen Spielbaum, dessen am weitesten links

verlaufender Pfad den Masterplan P darstellt. Wenn eine Störung auftritt, sind wir gezwungen, diesen Plan zu verlassen, können aber hoffentlich bei einem Knoten v wieder in ihn zurückkehren. Der fette Pfad startend von Knoten v besteht aus Knoten mit denselben Zuständen wie im Masterplan. Am Knoten v haben wir somit Kosten für den Pfad von der Wurzel r nach v, gegeben durch f(r,…,v) und erwartete zukünftige Kosten, gegeben durch g(s(v)). Wenn wir dem Masterplan die ganze Zeit über gefolgt wären, hätten wir dieselben erwarteten zukünftigen Kosten zu verbuchen, jedoch geringere Pfadkosten. Typischerweise ist der einzige Knoten mit erwarteten Gesamtkosten Null das Blatt p des Masterplans.

4.3.4.4 Experimente mit heuristischer Spielbaumsuche

Im Folgenden beschreibe ich zwei Reparaturprogramme, so genannte Reparaturengines, die miteinander vergleichen wurden. Die erste Engine repariert einen Plan nach einer Störung unter der Annahme, dass keine weiteren Störungen auftreten werden. Sie benutzt dazu ein zum deterministischen Fleet-Assignment-Problem leicht modifiziertes Time-Space Netzwerk, so dass die Lösung des dazugehörigen Flussproblems optimal unter der Annahme ist, dass keine weiteren Störungen auftreten werden. Die zweite Engine, die auf dem Reparaturspiel aufbaut, vergleicht verschiedene Lösungen des modifizierten Time-Space-Netzwerk-Flussproblems und untersucht verschiedene Szenarien, die in nächster Zukunft entstehen können.

Der „kurzsichtige MIP"-Löser ist ein einfaches, aber nahe liegendes Reparaturwerkzeug. Er stellt nach einer Störung ein gemischt-ganzzahliges lineares Programm auf, das ähnlich zu demjenigen ist, mit dem auch das ursprüngliche Fleet Assignment und der ursprüngliche Rotationsplan erzeugt wurden. Man braucht lediglich von einem Startzeitpunkt T auszugehen und einige Nebenbedingungen für die gestörten Flüge zu verändern [12][17], sowie die Kostenfunktion abzuändern, weil die Reparaturkosten hauptsächlich von den Veränderungen am Plan verursacht werden. Wir modellieren das Problem, eine Reparaturlösung zu finden, als ganzzahliges Min-Cost-Multi-Commodity-Flussproblem auf einem erweiterten Time-Space Netzwerk. Wir fügen die Besonderheiten von Verspätungen und Ausfällen von Flügen zu dem Modell hinzu.
In einem MIP, das zu diesem modifizierten Time-Space Netzwerk gehört, gibt es für jeden Flug l und für jede Flotte f eine binäre Entscheidungsvariable $x_{l,f}$, die anzeigt, ob Flug l von Flugzeugtyp f geflogen werden soll. In einer gültigen Lösung ist $x_{l,f}$ genau dann gleich 1, wenn Flug l von Flugzeugtyp f geflogen wird. Ausfälle von Flügen können eingebracht werden, indem für jeden Flug l eine zusätzliche Entscheidungsvariable $x_{l,*}$ eingeführt wird, die genau dann 1 wird, wenn Flug l gelöscht werden soll.
Die Fähigkeit, Verspätungen zu modellieren, geschieht auf die folgende Weise:Zunächst schränken wir die Freiheitsgrade, wie ein Flug verzögert werden kann, auf eine kleine Anzahl von D vielen möglichen Verspätungen ein; sagen wir auf 0, 30, 60 und 120 Minuten (D=4). Dann führen wir anstelle der einen Variable $x_{l,f}$ für eine Flotte f, die Flug l bedienen kann, D viele Variablen ein, jeweils eine für jede Verspätungsart. Die beschriebenen Erweiterungen lassen sich problemlos in das Time-Space Netzwerk einfügen. Darüber hinaus erlauben uns diese Variablen, direkt die Kosten einer Reparaturalternative zu minimieren, indem wir sie in die Kostenfunktion mit einbringen. Wir zählen dort die Anzahl von Verspätungen, Neuzuweisungen von Flotten und Ausfälle.
Auf diese Weise werden Reparaturkosten minimiert, und der resultierende Plan ist optimal unter der Annahme, dass keine weiteren Störungen auftreten werden. Nichtsdestotrotz ist

diese Engine ein kurzsichtiger Löser innerhalb des von uns angestrebten dynamischen Modells.

„T3“ ist ein Löser, der das Reparaturspiel mit Hilfe eines vorausschauenden Suchalgorithmus spielt. Eine Vorausschauprozedur macht sich die Dynamik des Problems lokal um einen Zeitpunk T und T+d folgendermaßen zu Nutze: Anstatt lediglich eine kurzsichtige optimale Lösung für die Reparatur zu erzeugen, generiert sie gleich mehrere. Diese werden unsere *möglichen Züge* genannt. Nur für diese Züge untersuchen wir, welche relevanten Störungen in den nächsten d Minuten auftreten können.
In all den entstehenden Szenarien reparieren wir den Plan mit Hilfe des oben beschriebenen kurzsichtigen MIP-Lösers, der uns Schätzwerte für die Szenarien liefert. Im Moment experimentieren wir nur mit dieser Tiefe-3-Suche. Wir gewichten die Szenarien bzgl. ihrer Eintrittswahrscheinlichkeiten und maximieren an der Wurzel über die Erwartungswerte der Szenarien. Bezüglich des neuen Übergangsplans haben wir nun nicht die Kosten über erwartete Inputdaten maximiert, sondern näherungsweise die erwarteten Kosten über die möglichen Szenarien optimiert.
Seien nun U und L untere und die obere Schranken aller möglichen Werte, die das Spiel annehmen kann. Der Suchalgorithmus, den wir verwenden, ist ein Alphabeta-Algorithmus, der durch Ersetzen des Min-Spielers durch einen Avg-Spieler modifiziert wurde. Wenn die Natur am Zuge ist, bekommt jeder Zug der Natur ein Gewicht, und die Werte der Nachfolger werden bzgl. ihrer Gewichte summiert. Wenn wir uns also an einem Naturknoten befinden und die Untersuchung der ersten Nachfolger dazu geführt hat, dass auch im Extremfall der Wert des Knotens nicht mehr unter eine Schranke β fallen kann, haben wir einen Cutoff. Analoges gilt natürlich auch für die α-Schranke. Diese Überlegungen führen zu folgendem einfachen Suchalgorithmus:

Wert wmaxav(Knoten v, Wert α, Wert β)
 generiere alle Nachfolger $v_1,\ldots,v_b$ von v
 val := 0;
 if b = 0 **return** h(v) // ein Blatt
 for i := 1 **to** b
  if v ist MAX-Knoten
   α := wmaxav(α,wmaxav(v i,α,β));
   if α≥β **or** i = b **return** α;
  else // seien $w_1,\ldots,w_b$ die Gewichte der Knoten $v_1,\ldots,v_b$
   α' := max (L, (α – val – U · $\sum_{j=i+1}^{b} w_j$)/ w_i)
   β' := min (U, (β – val – L · $\sum_{j=i+1}^{b} w_j$) / w_i)
   val += wmaxav(v_i,α',β') · w_i;
   if val + L · $\sum_{j=i+1}^{b} w_j \geq \beta$ **return** β;
   if val + U · $\sum_{j=i+1}^{b} w_j \leq \alpha$ **return** α;
   if i = b **return** val;

Der Zuggenerator für den Max-Spieler erzeugt mehrere „gute“, aber unterschiedliche Reparaturalternativen für einen gestörten Plan. Sie führen alle so schnell und kostengünstig wie möglich zurück in den Masterplan. Dabei kann jede Alternative aus einer Mehrzahl von Reparaturoperationen bestehen: aus Start- und Landezeitverschiebungen, aus Neuzuweisungen von Flügen auf Flotten und aus Streichungen von Flügen. Als Basis benutzen wir dabei das kurzsichtige MIP des vorangegangenen Abschnitts 3.1, um Reparaturalternativen zu erzeugen. Wir modifizieren jedoch die Branch&Bound-Suche des MIP-Lösers. Wenn eine neue ganzzahlige Lösung (also in unseren Worten eine gute und gültige Reparaturalternative) gefunden wurde, die außerdem noch nah am kurzsichtigen

Optimum (z.B. nur 1% weit weg) liegt, wird diese Lösung gespeichert und durch Hinzufügen geeigneter Cuts ungültig gemacht. Die Branch&Bound-Suche terminiert, sobald c viele Reparaturalternativen gefunden wurden. In [K13] verwenden wir c=3.

Um auch wirklich unterschiedliche Lösungsvorschläge zu bekommen, muss man die zusätzlichen Cuts sorgfältig auswählen. Wir berücksichtigen für Cuts nur Entscheidungsvariablen von Flügen, die anders gehandhabt werden (verspätet, neu zugewiesen oder gestrichen) als im Originalplan. Wir gruppieren diese Variablen nach Flugzeugen. Für jedes Flugzeug, das k>0 geänderte Flüge bedient, fügen wir einen Schnitt hinzu, der alle Entscheidungsvariablen der k geänderten Flüge (für die Flotte, zu der das Flugzeug gehört) zusammenaddiert. Dann schränken wir diese Summe auf höchstens k-1 ein. Man beachte, dass diese Summe in der aktuellen ganzzahligen Lösung gleich k ist. Deshalb schneidet diese Ungleichung auch die aktuelle ganzzahlige Lösung aus dem Lösungsraum des MIP heraus.

Der Zuggenerator für den AVG-Spieler untersucht an AVG-Knoten, welche Abflüge in den nächsten 15 Minuten anstehen. Die atomaren Störungen „Streichung“ und „Verspätung von 30, 60 und 120 Minuten“ werden für jeden Flug, der in diesem Zeitraum den Boden verlässt, generiert. Die atomaren Störungen werden gemäß ihren (vorgegebenen) Eintrittswahrscheinlichkeiten gewichtet. Im Prinzip würden wir jetzt gerne alle diese atomaren Störungen sowie alle ihre Kombinationen untersuchen. Leider führt dies zu einer zu großen Anzahl möglicher Szenarien. Zurzeit beschränken wir daher die Suchbreite dadurch, dass wir nur atomare Störungen betrachten.

Experimentelle Ergebnisse zeigen, dass es tatsächlich möglich ist, mittels heuristischer Spielbaumsuche den kurzsichtigen Löser im Fleet-Assignment-Reparaturspiel statistisch signifikant zu schlagen.
Wir haben aus einem realen Flugplan der Lufthansa mit 144 Flugzeugen in 6 Flotten mit 20603 Flugsegmenten 14 von 21 aufeinander folgende Tage herausgenommen und daraus 14 unabhängige Teilpläne als Testinstanzen generiert, für jeden der 14 Tage eine Instanz. Wie oben bereits erwähnt, wird dabei jede Instanz in 15 Minuten-Intervalle diskretisiert und alle Ereignisse innerhalb eines Intervalls werden als gleichzeitig stattfindend angenommen. Der Simulator kann verschiedene Reparaturengines zur Störungsbehebung einsetzen, und so haben wir das Verhalten des kursichtigen MIP-Lösers mit dem der T3-Engine verglichen.

Als Bewertung der Simulationsergebnisse haben wir vier relevante Maße während der Simulation protokolliert und über eine gewichtete Kostenfunktion zu unseren Reparaturkosten umgerechnet. Als Maße kamen dabei die Summe der benötigten Startverzögerungen in Minuten (TIM), die Anzahl der Wechsel eines Flugzeugtyps für einen Flug (ECH) und die Anzahl der Flug-Streichungen (CNL) zum Einsatz. Zusätzlich haben wir noch die Änderungen der Erlöse, die sich aus verloren gegangenen Passagieren ergeben, berücksichtigt. In Gesprächen mit unserem Kooperationspartner haben wir die folgende Kostenfunktion vereinbart.
c(TIM,ECH,CNL,Erlöse) = 50 TIM + 10000 ECH + 100000 CNL beschreiben die Kosten, die TIM viele Verzögerungen, ECH-viele Flugzeugtypenwechsel und CNL-viele Flugstreichungen erzeugen. Falls ein Löser zu einem Zeitpunkt keine Lösung findet, soll dies mit 400000 Kosteneinheiten bestraft werden. Seien die Wahrscheinlichkeiten für Störungen 0.003 für Flugstreichungen, 0.04 für Verspätungen um 120 Minuten, 0.16 für Verspätungen um 60 Minuten und 0.24 für Verspätungen um 30 Minuten. Dann sieht ein typischer Testlauf wie in Tabelle 2 dargestellt aus, wobei es primär um die Frage ging, ob man für das gegebene

Problem unter halbwegs realistischen Annahmen überhaupt Heuristiken finden kann, die eine deterministische rollierende Planung schlagen.

Die Spalten 2 bis 6 von Tabelle 1 zeigen die Ergebnisse des kurzsichtigen MIP-Lösers, die Spalten 7 bis 11 gehören zu der T3-Engine. Die Δ-Spalte zeigt die Kostendifferenz der c-Spalten der zwei Reparaturengines; positive Werte bedeuten, dass die T3-Engine weniger Reparaturkosten verursacht hat als der kurzsichtige Löser. Wie aus der Tabelle ersichtlich wird, gewinnt die T3-Engine 10 von 14 Tagen gegenüber dem kurzsichtigen Löser. Nicht abgebildet ist, dass beide Löser dreimal keine Lösung produzieren konnten. Über alle 14 Tage zusammen kann die T3-Engine 3% Reparaturkosten einsparen. Ohne Berücksichtigung der nicht gelösten Aufgaben sind es sogar 3.32%.

Für jeden Reparaturschritt eines 15 Minuten-Intervalls benötigt der MIP-Löser ungefähr 8 Sekunden. Der T3-Löser ist ungefähr 200-mal langsamer, da er im Mittel 213 Suchknoten in jedem Schritt untersuchen muss. Eine Parallelisierung mittels dynamischer Lastverteilung brachte auch den T3-Löser unter die wichtige Realzeitgrenze [K14].

Tabelle 1. Simulationsergebnisse für Wahrscheinlichkeiten 0.003/0.04/0.16/0.24

Datum	MIP					T3					
	TIM	CNL	ECH	Erlöse	c	TIM	CNL	ECH	Erlöse	c	Δ
01/03	12210	8	2	-684	1431184	12120	8	2	-11374	1437374	-6190
01/04	11460	6	1	2028	1180972	11550	4	0	1145	976355	204617
01/05	10950	6	4	-24978	1212478	11340	8	5	-20407	1437407	-224929
01/06	13830	8	1	-5654	1507154	13470	8	1	-2567	1486067	21087
01/07	10530	7	2	-28385	1274885	10950	7	3	-33248	1310748	-35863
01/08	6000	4	4	-49501	789501	5430	2	4	-49784	561284	228217
01/09	11640	4	2	-29410	1031410	11570	4	2	-20977	1019477	6933
01/10	11730	6	3	-20403	1236903	11130	6	3	-25713	1212213	24690
01/11	12060	8	0	2003	1400997	12150	6	3	12119	1225381	175616
01/12	12030	6	2	-1971	1223471	11790	8	2	974	1408526	-185055
01/13	12630	8	0	580	1430920	12570	6	1	1036	1237464	193456
01/14	10410	6	4	-38488	1198988	10020	6	4	-36133	1177133	21855
01/15	5790	2	0	-1000	490500	5760	2	0	-1000	489000	1500
01/16	12270	5	0	-133	1113633	12090	4	0	257	1004243	109390
Σ	153540	84	25	-195996	16522996	152040	79	30	-185672	15987672	535324

Obwohl an jedem Simulationstag fast 100 Entscheidungen getroffen werden, konnten wir die einzelnen Entscheidungen nicht direkt zur Klärung von Signifikanzfragen zu Hilfe nehmen, da die einzelnen Entscheidungen nicht unabhängig voneinander sind. Innerhalb eines Tages sind die Entscheidungen Folgeentscheidungen von Folgeentscheidungen…
Die Resultate von einzelnen Tagen sind ebenfalls mit Vorsicht zu genießen. Zum einen scheinen sie nicht einer Normalverteilung zu genügen, zum anderen sind sie auch von der Struktur des Plans an einem bestimmten Tag abgängig. Wir nehmen daher Durchschnittswerte mehrerer Tagesdaten als Messpunkte.

Tabelle 2. Durchschnittlich eingesparte Tagesreparaturkosten der T3-Engine gegenüber dem MIP-Löser

	Lauf 1	Lauf 2	Lauf 3	Lauf 4	Lauf 5
Woche 1	70293	27696	32261	-9238	-15799
Woche 2	8389	48778	11580	-1253	9144

Wir haben deshalb zusätzliche Testläufe mit den bereits oben verwendeten Störungswahrscheinlichkeiten 0.003/0.04/0.16/0.24 durchgeführt. Dabei kam jeweils ein anderer Random-Seed zum Einsatz. Als Messpunkte wurden die durchschnittlichen Tagesreparaturkosten über eine komplette Woche genommen, wobei wir davon ausgehen, dass sich die beiden Wochen des Plans „hinreichend wenig" in ihrer Struktur unterscheiden. Damit sind strukturelle Einflüsse (außer denjenigen, die auf die Pseudo-Zufälligkeit des Zufallsgenerators zurückzuführen sind) eliminiert, und wir erhalten insgesamt 10 Messpunkte, die in Tabelle 2 angegeben sind.
Ein Eintrag Woche i / Lauf j beschreibt den durchschnittlichen absoluten Tagesgewinn bzw. Verlust des T3 Verfahrens gegenüber dem MIP Verfahren in Woche i bei Simulationslauf j. Positive Werte sind günstig für das T3-Verfahren. Der Mittelwert über diese Werte beträgt 18185 Einheiten, die Standardabweichung 26762. Gehen wir nun davon aus, dass die gegebenen Durchschnittswerte annähernd normalverteilt sind, ergibt sich mit Hilfe der t-Verteilung, dass mit 95% Sicherheit die T3-Engine besser ist als die Myopic-MIP-Engine.

Fazit: Durch Spielen des Reparaturspiels wurden robustere (Teil-)Pläne für die Flugzeugumlaufplanung erzeugt als mit einem kurzsichtigen MIP-Löser. Unser vorausschauender Reparaturalgorithmus schlägt einen kurzsichtigen MIP-Löser statistisch signifikant in der vereinbarten Simulationsumgebung.

5 Liste der Veröffentlichungen

5.1 Konferenzen

[K16] C. Donninger, U. Lorenz. Innovative Opening-Book Handling. *Proc. Advances in Computer Games (ACC) 11*, 2005, Taipei, Taiwan, to appear

[K15] U. Lorenz, T. Tscheuschner. Player Modeling, Search Algorithms and Strategies in Multi Player Games. *Proc. Advances in Computer Games (ACG) 11*, 2005, Taipei, Taiwan, to appear

[K14] J. Ehrhoff, S. Grothklags, U. Lorenz. Parallelism for Perturbation Management, Euro-Par 2005, to appear

[K13] J. Ehrhoff, S. Grothklags, U. Lorenz. Störungsmanagement und Planung unter Unsicherheiten, angewendet auf die Flugplanung. *GOR: Entscheidungsunterstützende Systeme in Supply Chain Management und Logistik*. Physica Verlag, pp. 335 - 356

[K12] C. Donninger, U. Lorenz. The Chess Monster Hydra. *Proc. of 14th International Conference on Field-Programmable Logic and Applications (FPL)*, 2004, Antwerp – Belgium, LNCS 3203, pp. 927 – 932, eds. J. Becker, M. Platzner, S. Vernalde

[K11] R. Elsässer, U. Lorenz, T. Sauerwald. Agent-Based Information Handling in Large Networks. *Proc. of the 29th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2004, Prag, LNCS 3153, pp. 586-598

[K10] U. Lorenz. Beyond Optimal Play in Two-person-Zerosum Games. *Proc. of the 12th Annual European Symposium on Algorithms (ESA)*, 2004, Bergen – Norway, LNCS 3221, pp. 749 – 759, eds. S. Albers and T. Radzik

[K9] C. Donninger, A. Kure, U. Lorenz. Parallel Brutus: The First Distributed, FPGA Accelerated Chess Program. *Proc. of 18th International Parallel & Distributed Processing Symposium (IPDPS)*, 2004, Santa Fe – NM USA, IEEE Computer Society

[K8] S. Grothklags, U. Lorenz, T. Sauerwald. Experiments with the Repair Game. *Aviation Application Cluster at Institute for Operations Research and the Management Sciences (INFORMS)*, 2004, Denver – CO USA,

[K7] J. Ehrhoff, S. Grothklags, J. Halbsgut, U. Lorenz, T. Sauerwald. The Repair Game: Robust Plans and Disturbation Management in Aircraft Scheduling by the Help of Game Tree Search. *Proc. of 43rd Annual Symposium of the Airline Group of the International Federation of Operational Research Societies (AGIFORS)*, 2003, Paris, France

[K6] U. Lorenz, B. Monien. The Secret of Selective Game Tree Search, when Using Random-Error Evaluations. *Proc. of 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, 2002, Antibes – Juan les Pins, France, LNCS 2285, pp. 203-124, eds. H. Alt and A. Ferreira

[K5] U. Lorenz. Parallel Controlled Conspiracy Number Search. *Proc. of the 8th International Euro-Par Conference (Euro-Par),* 2002, Paderborn. Germany, LNCS 2400, pp. 420-431, eds. B. Monien and R. Feldmann

[K4] U. Lorenz. Parallel Controlled Conspiracy Number Search. *Proc. of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001, Crete – Greece, pp. 320-321

[K3] U. Lorenz. Controlled Conspiracy-2 Search. *Proc. of the 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, 2000, Lille-France, LNCS 1770, pp. 446-478, eds. Reichel and Tison

[K2] U. Lorenz, V. Rottmann. Parallel Controlled Conspiracy Number Search. *Proc. Advances in Computer Chess (ACC) 8*, 1997, Maastricht, Netherlands, pp. 129-152, eds. J. van den Herik and J.W.H.M. Uiterwijk

[K1] I. Althöfer, C. Donninger, U. Lorenz, V. Rottmann. On Timing, Permanent Brain and Human Intervention. *Proc. Advances in Computer Chess (ACC) 7*, ed. J. van den Herik

5.2 Eingeladene Vorträge

[V1] J. Ehrhoff, S. Grothklags, U. Lorenz, T. Sauerwald. The Repair-Game: Disruption Management and Robust Planning in Aircraft Scheduling with the Help of Game Tree Search. Lufthansa Systems. *NetLine Forum* 2004, Potsdam – Germany

5.3 Zeitschriften

[Z6] B. Doerr and U. Lorenz. Error Propagation in Game Trees. Mathematical Methods of Operations Research
Status: akzeptiert

[Z5] C. Donninger and U. Lorenz. The Hydra Project. *XilinX Xcell Online* (ausgewählt). Issue 53, 2005

[Z4] C. Donninger and U. Lorenz. The Hydra Project. *XilinX Xcell Journal* . Issue 53, pp. 94-97, 2005

[Z3] R. Elsässer, U. Lorenz, T. Sauerwald. Agent-Based Information Handling in Large Networks. *Discrete Applied Mathematics*
Status: akzeptiert

[Z2] U. Lorenz and B. Monien. Error Analysis in Minimax Trees. *Journal of Theoretical Computer Science (TCS)* 313, 2004, Elsevier, pp. 485-498

[Z1] U. Lorenz, V. Rottmann. Controlled Conspiracy Number Search. *International Computer Chess Association (ICCA) Journal*, Vol. 18, No. 3, 1995, pp 135-147

5.4 Qualifizierende Arbeiten

[Q2] U. Lorenz.
Controlled Conspiracy Number Search.
Dissertation (in Deutsch), Universität-GH Paderborn, 2001

[Q1] U. Lorenz.
Controlled Conspiracy Number Search.
Diplomarbeit (in Deutsch), Universität-GH Paderborn, 1995

6 Fremdliteratur

[1] I. Althöfer I Root evaluation errors: how they arise and propagate. *ICCA Journal* 11: 55–63, 1988

[2] B.W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence* 21: 327–350, 1983

[3] Y. Björnsson and T. Marsland. Multi-Cut Alphabeta-Pruning in Game_Tree Search. *Theoretical Computer Science*, vol. 252(1-2), pp. 177-196

[4] G. Carl, T. Gesing. Flugplanung als Instrument des Informationsmanagements zu Ressourcenplanung und -steuerung einer Linienfluggesellschaft. In: *Daduna JR, Voß S (Hrsg) Informationsmanagement im Verkehr*. Springer, Heidelberg, S 167–198, 2000

[5] J.H. Condon and K. Thompson. Belle Chess Hardware. *Proc. of Advances in Computer Chess 3*, Pergamon Press, pp. 44-54, 1982

[6] R.L. Daniels, P. Kouvelis. Robust scheduling to hedge against processing time uncertainty in single-stage production. *Management Science* 41: 363–376, 1995

[7] C. Donninger. Null move and deep search. *ICCA Journal*, 16(3):137-143, 1993

[8] S. Engell, A. Märkert, G. Sand, R. Schultz. Production planning in a multiproduct batch plant under uncertainty. Preprint 495-2001, FB Mathematik, Gerhard-Mercator-Universität Duisburg, 2001

[9] R. Feldmann. Spielbaumsuche mit massiv parallelen Systemen. Doctorial-Thesis, University of Paderborn, Germany

[10] S. Grothklags. Fleet assignment with connection dependent ground times. *Proceedings of the 11th Annual European Symposium on Algorithms* (ESA 2003), Budapest, S 667–678, 2003

[11] Z. Gu, E.L. Johnson, G.L. Nemhauser, Y. Wang. Some properties of the fleet assignment problem. *Operations Research Letters 15*: 59–71, 1994

[12] C.A. Hane, C. Barnhart, E.L. Johnson, R.E. Marsten, G.L. Nemhauser, G. Sigismondi. The fleet assignment problem: solving a large-scale integer program. *Mathematical Programming 70*: 211–232, 1995

[13] E. Horvitz, J. Breese, M. Henrion. Decision theory in expert systems and artificial intelligence. *Journal of Approximate Reasoning, Special Issue on Uncertainty in Artificial Intelligence*, S 247-302, 1988

[14] F-H. Hsu. IBM's Deep Blue Chess Grandmaster Chips. *IEEE Micro*, 18(2):70-80, 1999

[15] F-H. Hsu, T.S. Anantharaman, M.S. Campbell, and A. Nowatzyk. *Computers, Chess, and Cognition*, Chapter 5, Deep Thought, pp. 55-78, Springer, 1990

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

[17] A.I. Jarrah, J. Goodstein, R. Narasimhan. An efficient airline re-fleeting model for the incremental modification of planned fleet assignments. *Transportation* Science 34: 349–363, 2000

[18] H. Kaindl and A. Scheucher. The Reason fort he benefits of Minmax Search. In *Proc. of the 11th IJCAI*, pp. 322-327, Detroit, MI, 1989

[19] O. Kao. Parallel and Distributed Methods for Image Retrieval with dynamic Feature Extraction on Cluster Architectures. *12th International Workshop on Database and Expert Systems Applications*, 2001

[20] D.E. Knuth, R.W. Moore An analysis of alpha-beta pruning. *Artificial Intelligence 6*: 293–326, 1975

[21] P. Kouvelis, R.L. Daniels, G. Vairaktarakis. Robust scheduling of a two-machine flow shop with uncertain processing times. *IIE Transactions 32*: 421–432, 2000

[22] V. J. Leon, S.D. Wu, R.H. Storer. A game-theoretic control approach for job shops in the presence of disruptions. *International Journal of Production Research 32*: 1451–1476, 1994

[23] Lexikon der Mathematik. Bd. 3 Inp bis Mon, Spektrum Akademischer Verlag, Berlin, 2001

[24] U. Lorenz and B. Monien. The Secret of Selective Game Tree Search, when Using Random-Error Evaluations. *Proc. of the 19th STACS* (H. Alt, A. Ferreira eds), Springer LNCS, pp. 203-214, 2002

[25] A. Mas-Colell, M.D. Whinston, and J.R. Green. Microeconomic Theory. Oxford University Press, 1995

[26] J. McCarthy in Computers, Chess and Cognition, eds. T.A. Marsland and J. Schaeffer, 1990, ISBN 0-387-97415-6

[27] R.H. Möhring, A.S. Schulz, M. Uetz. *Approximation in Stochastic Scheduling: The Power of LP-Based Priority Schedules*. Journal of ACM, Vol. 46, No. 6, pp. 924-942, 1999

[28] J.M. Mulvey, R.J. Vanderbei, S.A. Zenios. Robust optimization of large-scale systems. *Operations Research 43*: 264–281, 1995

[29] D.S. Nau. Quality of decision versus depth of search on game trees. PhD Thesis, Duke University, Durham, 1979

[30] C.H. Papadimitriou. Games against Nature. *Journal of Computer and System Science* 31, pp 288-301, 1985

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

[32] A. Reinefeld. An improvement of the Scout tree search algorithm. *ICCA Journal 6*: 4–14, 1983

[33] W. Römisch, R. Schultz. Multistage stochastic integer programming: An introduction. In: Grötschel M, Krumke SO, Rambau J (eds) *Online Optimization of Large Scale Systems*, Springer, Berlin Heidelberg New York, S 579–598, 2001

[34] J.M. Rosenberger. Topics in Airline Operations. PhD Thesis, Georgia Institute of Technology, Atlanta, 2000

[35] J.M. Rosenberger, E.L. Johnson, G.L. Nemhauser. Rerouting aircraft for airline recovery. TLI-LEC 01-04 Georgia Institute of Technology, 2001

[36] J.M. Rosenberger, A.J. Schaefer, D. Goldsman, E.L. Johnson, A.J. Kleywegt, G.L. Nemhauser. Simair: A stochastic model of airline operations. *Winter Simulation Conference Proceedings*, 2000

[37] S. Russel, P. Norvig. Artificial intelligence: A modern approach. Prentice Hall, Upper Saddle River, 1995

[38] D.B. Shmoys and C. Swamy. Stochastic Optimization is (almost) as easy as deterministic Optimization. *In Proc. of 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 228-237, 2004.

[39] D.J. Slate and L.R. Atkin. Chess 4.5 – The Northwestern University Chess Program. *Chess Skill in Man and Machine, P.W. Frey (ed)*, Springer, pp. 82-118, 1977

[40] T.J. Schaefer. The complexity of some two-person perfect-information games, *J. Computer System Science* 16 , pp. 185-225, 1978

[41] A. Scholl. Robuste Planung und Optimierung: Grundlangen, Konzepte und Methoden, Experimentelle Untersuchungen. Springer, Berlin Heidelberg New York, 2001

[42] G. Schrüfer. Presence and Absence of Pathology on Game Trees. *Proc. of Advances in Computer Chess 4*, pp. 101-112. Pergamon Press, 1986

[43] L. Suhl, T. Mellouli. Optimierungssysteme. Springer-Verlag, 2005, ISBN 3-540-26119-2

[44] D. Wagner. Algorithms and Models for Railway Optimization. In Proceedings of 8th Workshop on Algorithms and Data Structures (WADS'03), LNCS 2748, pp. 198-206, 2003

[45] S. Mecke, D. Wagner. Solving Geometric Covering Problems by Data Reduction. *Proc. of the 12th Annual European Symposium on Algorithms (ESA)*, 2004, Bergen – Norway, LNCS 3221, pp. 760 – 771, eds. S. Albers and T. Radzik

7 Lehre

7.1 Lehrveranstaltungen

- Übung *C und Unix Einführungskurs*
- Übung *Datenbanken I*, WS 2000/01
- Übung *Parallele Algorithmen II*, SS 2001
- Übung *Algorithmische Spieltheorie*, SS 2002
- Vorlesung *Parallele Programme mit C/C++ und MPI*, SS 2002, SS 2003
- Übung *Datenstrukturen und Algorithmen*, SS 2003
- Übung *Verteilte Systeme*
- Vorlesung *Effiziente Nutzung paralleler Systeme mittels Message Passing*, SS 2004, SS 2005
- Übung *Einführung in Berechenbarkeit, Komplexität und formale Sprachen*, WS 2004/05
- Übung *Graphentheorie und Rechnernetze II*, WS 2004/05

7.2 Betreute Diplomarbeiten, Studienarbeiten und Promotionen

- Jörg Halbsgut: *Maschinelle Lernverfahren zur personenbezogenen Klassifizierung von TV-Metadaten* (Studienarbeit, Erstgutachter)
- Tobias Tscheuschner: *Spielermodellierung, Suchalgorithmen und Spielstrategien in Mehrpersonen-Nullsummen-Spielen* (Studienarbeit, Erstgutachter)
- Tobias Tscheuschner: *Intelligente Vorausschau zur Behandlung von Mehrpersonen-Spielen: Spielermodelle und Algorithmen im Vergleich* (Betreuung der Diplomarbeit zusammen mit Prof. Monien)
- Thomas Sauerwald: *Randomisiertes Broadcasting auf großen Netzwerken mit guten Topologieeigenschaften* (Betreuung der Diplomarbeit zusammen mit Junior Prof. Dr. Elsässer)
- Sven Grothklags:
  (Betreuung der Dissertation zusammen mit Prof. Monien)

8 Mitarbeit in Projekten

- Mitarbeit im DFG-Schwerpunktprogramm
  Efficient Algorithms for Discrete Problems and their Applications
  Teilprojekt: Sequential and Distributed Selective Tree-Search Algorithms

- Mitarbeit im EU-Projekt
  Ubiquitious Personalized Interactive Multimedia TV-Systems and Services (UPTV)

- Mitarbeit im EU-Projekt
  Algorithms and Complexity Future Technologies (ALCOM-FT)
  Work Package 2 – Networks and Communication
  Work Package 4 – generic Methods

- Mitarbeit im EU-Projekt
  Dynamically Evolving, Large Scale Information Systems (DELIS)
  Teilprojekt 3

- Mitarbeit im DFG-Sonderforschungsbereich 614
  Selbstoptimierende Systeme des Maschinenbaus
  Teilprojekt A1 – Modellorientierte Selbstoptimierung

- Initiierung und Mitarbeit im Industrieprojekt *Hydra*
  finanziert von PAL Computer Systems, Abu Dhabi, UAE

Im Rahmen obiger Projekte:

- Entwurf und Analyse zur Informationsverteilung in großen Netzwerken mit Hilfe sich zufällig im Netz bewegender Agenten. [K11][Z3]

- Entwurf und Analyse selektiver Spielbaumsuchalgorithmen
  [K2][K3][K4][K5][Z1][Z2]

- Untersuchung des Potentials von FPGA-Technologie für Algorithmen
  [K9][K12][K16][Z4][Z5]

- Modellierung und Algorithmen für robuste Planung [K13][K14]