Eine allgemeine selbstlernende Strategie für nicht-kooperative Spiele

- im KI-Framework jGameAI -

Fachbereich Mathematik und Informatik
Arbeitsgruppe Künstliche Intelligenz
der Freien Universität Berlin

Autor: Maro Bader
Betreuer: Prof. Dr. Raúl Rojas, Marco Block-Berlitz

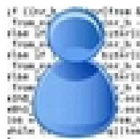
9. Dezember 2008

Zusammenfassung

Unter nicht-kooperative Spiele fallen die meisten Gesellschafts- und Brettspiele, bei denen Spieler versuchen, eine siegbringende Position durch den Vergleich zukünftiger Spielstellungen zu erreichen. Auch viele Situationen des Alltags, etwa in Wirtschaftsprozessen und Verhandlungssituationen lassen sich mit spieltheoretischen Formalismen gut beschreiben und gegebenenfalls lösen.

Starke Spieler entstehen, wenn man aus gespielten Partien sinnvolle Rückschlüsse für zukünftiges Verhalten gewinnen kann. Dies kann mit maschinellem Lernen bewerkstelligt werden. Die Konzepte einer verallgemeinerten selbstlernenden Computerstrategie für eine breite Klasse an Spielen werden in dieser Diplomarbeit untersucht und vorgestellt.

Die entwickelten Such- und Lernalgorithmen $UPP{-}max^n$ und $TD{-}Prob^n(\lambda)$ wurden im KI-Framework *jGameAI* eingebaut und finden ihre Verwendung bereits in mehreren Computerspielen für gängige Spiele. Das Framework wurde eigens für diese Diplomarbeit realisiert und wird von der AG Spieleprogrammierung der Freien Universität ständig erweitert.



https://sourceforge.net/projects/jgameai/

Inhaltsverzeichnis

1 Einleitung und Motivation **5**

2 Verwandte Arbeiten **10**

2.1 Komplexität von Spielen 10

2.2 Suchbaumbasierte Verfahren zur Lösung von Spielen 12

2.3 Klassifizierung von Lösungen 13

2.4 Einsatz von Lernmethoden 14

3 Abstraktion von Spielen **16**

3.1 Spiele als nicht deterministische Zustandsautomaten 16

3.1.1 Terminale Zustände 17

3.1.2 Zufallsereignisse und unvollständige Information 20

3.1.3 Spielbaum und Partien 22

3.2 Suchbaum und Strategie 22

3.2.1 Suchbaum 22

3.2.2 Strategie 23

3.2.2.1 Spekulative Strategien 25

3.2.2.2 Siegreiche Strategien 25

3.2.3 Strategiebaum und Suchbreite 26

3.3 Bewertung und Suche 27

3.3.1 Rationales Spielen durch Bewertungsfunktion 28

3.3.2 Ideale Bewertungsfunktion 29

3.3.3 Gewichtete Stellungskriterien 30

3.3.4 Verfeinerung durch Stellungsklassen 31

3.3.5 Normierung der Gewichtsvektoren 31

3.3.6 Auswahl- und Erwartungsfunktion 32

4 Ein allgemeiner Suchalgorithmus **35**
4.1 Das strategische Nash-Gleichgewicht 36
4.2 Suchalgorithmen für 2S-Nullsummenspiele 37
4.2.1 MiniMax-Algorithmus 37
4.2.2 Pruning-Methoden 38
4.2.3 Zugsortierungsheuristiken 40
4.3 Transpositionstabellen 42
4.3.1 Zobrist-Schlüssel 43
4.3.2 Varianten von Transpositionstabellen 43
4.4 Erweiterungen zu allgemeinen Spielklassen 45
4.4.1 Nicht-Nullsummenspiele 45
4.4.2 Tiefeniteration und Zeitmanagement 47
4.4.3 Entscheidungsvarianten der optimalen Strategie 48

5 Die MS-Spielsuche $UPP - max^n$ **50**
5.1 Das Mehrspieler-Entscheidungsproblem MES 50
5.2 Mehrspieler-Suchalgorithmen 52
5.2.1 Der max^n-Suchalgorithmus 52
5.2.2 Der *paranoid*-Algorithmus 54
5.3 Lösung des MES über die Entropie-Nutzenfunktion $U_{H'}(\overrightarrow{e})$ . . . 55
5.3.1 Notwendigkeit Spieleabhängiger Nutzenfunktionen 56
5.3.2 Die allgemeine Entropie-Nutzenfunktion $U_{H'}(\overrightarrow{e})$ 58
5.4 Der $UPP - max^n$-Suchalgorithmus 61
5.4.1 *paranoid*-Pruning im $UPP - max^n$-Suchalgorithmus . . . 63
5.4.2 Abschneidungen an Zufallsknoten in $UPP - max^n$ 64

6 Lernen mit $TD - Prob^n(\lambda)$ **67**
6.1 Maschinelles Lernen 68
6.2 $TD - Prob^n(\lambda)$ zur Optimierung der Gewichtskoeffizienten . . . 69
6.2.1 Konzept von $TD - Prob^n(\lambda)$ 70
6.2.2 Optimieren der Gewichtskoeffizienten 70
6.2.3 Diskussion und Ergebnisse 74

7 Beiträge und zukünftige Arbeiten **76**
7.1 Beiträge der Arbeit 76
7.2 Ausblick auf zukünftige Arbeiten 77
7.2.1 Zusätzliche Erweiterungen in $UPP - max^n$ einbauen . . . 77
7.2.2 Stellungskriterien selbstständig entdecken 78

7.2.3 Nicht-lineares Entfernungsmaß bei $U_{H'}$ 79
7.2.4 Risiko in Probabilistischen Spielen 80

8 Das Spieleframework jGameAI 83

8.1 Motivation und Zielsetzung 84
8.1.1 Projekt-Entwicklung 85
8.1.2 Übersicht Softwarearchitektur 85
8.2 Die Pakete 86
8.2.1 Die Basisklassen für alle Spiele und Spieler 86
8.2.1.1 Die Grundklasse für alle Spiele: Game 87
8.2.1.2 Die Zustandsverwaltung eines Spiels 87
8.2.1.3 Modellierung eines allgemeinen Spielers: Player . 89
8.2.1.4 Generische Bewertungsfunktion für Zustände . . 89
8.2.1.5 Graphische Darstellung von Spielen 89
8.2.2 Das interne Kommunikationssystem 90
8.2.3 Generische graphische Oberflächen 92
8.2.4 Suchen nach besten Zügen 94
8.2.5 Bewerten von Zügen durch die Computer-Spieler 96
8.2.6 Turniere und Trainingspartien 98
8.3 Aktuelle Projekte 100
8.3.1 Populäre Brettspiele: Schach, Dame und GO 100
8.3.2 Die nächste Herausforderung: Poker 100
8.3.3 Das Strategiespiel *Siedler von Catan* in jGameAI 102
8.3.4 Echtzeitanwendungen: *jBirds* und *jAsteroidsAI* 104
8.3.5 Die Spielwiese: 4-gewinnt!, 3-gewinnt! und 3-gewinnt!* . . 104
8.4 Ausblick und zukünftige Arbeiten 106

9 APPENDIX 108

9.1 Der $UPP - max^n$-Suchalgorithmus 108
9.2 Der $TD - prob^n(\lambda)$-Lernalgorithmus 112
9.3 Die Spiele 3Wins und 3WinsProb 116

Kapitel 1

Einleitung und Motivation

"Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better."
Elaine Rich, 1983

Möchte man die menschliche Intelligenz mit der künstlichen von Computern vergleichen, eignen sich Spiele dafür besonders gut. An die Computerspieler werden dabei hohe Ansprüche gestellt, die sie mit unterschiedlichen Methoden meistern müssen: rationale Spielhaltungen, schnelle Zugsuche und akkurate Einschätzung der Spielsituation. Ausgehend von menschlichen Konzepten können Computerspieler sich nach und nach einem Spiel anpassen und dieses erlernen.

Konrad Zuse, Claude Shannon, John von Neumann und Alan Turing gelten als die Pioniere der Informatik. Eine der ersten Anwendungen, mit denen sie sich (teilweise nur theoretisch) beschäftigten, waren Schachprogramme [Ertel 2008, Russel/Norvig 2004, Rojas 2002, Shannon 1950, Turing 1953]. Bereits damals stellten diese Forschungen ein interessantes Entwicklungsfeld dar, um Rechenleistung kombiniert mit künstlicher Intelligenz zu testen. Ein aus heutiger Sicht oft als unfair empfundener Wettkampf zwischen Mensch und Maschine begann, der auf immer mehr Spiele ausgeweitet wurde und zur ständigen Weiterentwicklung der Methoden künstlicher Intelligenz beigetragen hat.

Vor allem die Schachwelt profitierte viel von den immer stärker spielenden Computerprogrammen, da diese die eigenen theoretischen Arbeiten erweiterten und sich als gute Trainingspartner erwiesen. Schließlich wurde der Höhepunkt dieser Entwicklung mit dem von großem medialen Echo verfolgten Sieg

des Schachprogramms Deep Blue über den damaligen Weltmeister Kasparov erreicht [Campbell 2002]. Weitere siegreiche Computerprogramme folgten, so dass dieses Duell für uns Menschen mittlerweile als verloren gilt.

Einer der gängigen Vorwürfe an die Entwickler von Deep Blue war, dass ihr Programm zwar (fast) unbesiegbar sei, vom Spiel selbst aber wenig verstehe, kein Gefühl für das Spiel besitze und Partien nur aufgrund ungeheurer Rechenkraft gewinnen könne. Keiner der Parameter von Deep Blue war eigenständig erlernt, alle waren nur von Hand eingegeben [Russel/Norvig 2004]. Auch eines der gegenwärtig stärksten Schachprogramme, Hydra nutzt keine maschinelle Lernmethoden zur Weiterentwicklung [Donninger 2004].

Die Art und Weise, wie Computerspiele agieren, unterscheidet sich also noch sehr stark vom menschlichen Vorgehen. Ein Gefühl, gute Spielstellungen zu erlernen und aus vergangenen Partien, verlorenen wie gewonnenen, Rückschlüsse für zukünftige zu ziehen, wird in vielen verbreiteten Programmen nicht erreicht.der

Dabei besteht genau darin eine der Stärken menschlicher Spieler: sie erlernen siegreiches Spielen. Faszinierend daran ist, dass Menschen dadurch in der Lage sind, eine prinzipielle, in allen Spielen meisterliche Spielstärke zu erreichen. Offenbar reicht ein verallgemeinerter Such- und Lernansatz, der sich nach und nach an Gegebenheiten eines Spiels anpasst, dafür aus.

Es lohnt sich also, den Einsatz maschinellen Lernens in Spielen zu untersuchen, was bereits seit den Anfängen der Spieleprogrammierung unternommen wird [Samuel 1959]. Mittlerweile konnten dadurch starke Computerspieler entwickelt werden, die sogar den menschlichen Meistern überlegen sind. Für das Spiel Backgammon zeigte Gerald Tessauro 1995, dass dieser Ansatz zum Erfolg führt, denn sein Programm konnte den amtierenden Weltmeister mehrfach besiegen [Tesauro 1995].

Viele Herausforderungen konnten bisher allerdings noch nicht gemeistert werden. Poker beispielsweise erweist sich als interessantes Forschungsgebiet, da hier neue Elemente wie Zufall, Mehrspieler-Situation und Gegneranalyse hinzukommen. Die formale spieltheoretische Erweiterung für solche Spiele ist schnell vollbracht, eine sich durchsetzende allgemeine Lösung für die Bewältigung der zusätzlichen Komplexität scheint sich allerdings noch nicht abzuzeichnen.

Ein Erfolg versprechender Ansatz der künstlichen Intelligenz ist immer das Übernehmen von bewährten Lösungen natürlicher Intelligenz, hier etwa von uns Menchen, die auch starke und siegreiche Spieler hervorgebracht haben. Dabei muss ermittelt werden, auf welcher Entscheidungsgrundlage der menschliche Spieler seine Zugwahl trifft,

Wahrscheinlich noch wichtiger aber ist die Frage, wie genau Lernen etwa durch Spielpraxis oder Aneignung theoretischen Wissens stattfindet und wie dadurch die Spielstärke zunimmt. Sind diese Mechanismen einmal erkannt, können sie, kombiniert mit der ungeheuren Rechenkraft heutiger Computer, zu einem starken, möglichweise sogar unschlagbaren Spieler führen.

Eine interessante Frage ergibt sich, wenn man die grundlegende Herangehensweise von Menschen an neue Spiele untersucht. Auch ganz ohne Erfahrung schaffen es menschliche Spieler dennoch, gleich von Beginn an sinnvolle Züge zu finden und nach und nach ein Gespür für das Spiel zu entwickeln.

Ist es möglich, alle Spiele mit einer einzigen Suchmethode zu meistern und dabei nur einige wenige grundsätzliche Parameter an die spezifischen Rahmenbedingungen und Erfahrungen eines Spieles anzupassen? Eine universale Suche also, die mit maschinellem Lernen an die jeweiligen Spiele optimal angepasst werden kann?

Um diese Fragen bearbeiten zu können, wurde im Rahmen der AG-Spieleprogrammierung an der Freien Universität Berlin das KI-Framework *jGameAI* entwickelt, das eine verallgemeinerte Such- und Lernmethode anbietet. In dieser Arbeit werden die Entwicklung und Konzepte der Funktionen $UPP-Prob^n$ und $TD-Prob^n$ vorgestellt, die einen verallgemeinerten Lösungsansatz für viele Spiele darstellen. Einsatz und Erfolg beider Methoden sollen an einigen Beispielen vorgeführt werden.

Aufbau der Arbeit

Kapitel 2 beschäftigt sich mit verwandten Arbeiten und bekannten Computerprogrammen der Spieletheorie. Die Entwicklung in unterschiedlichen Spielen wird dargestellt. Dabei ist das Augenmerk auf diejenigen Lösungsansätze gerichtet, die maschinelles Lernen zum Verstehen eines Spiels und zur Verbesserung der Spielstärke verwenden. Die Komplexität von Spielen und eine allgemeine Klassifizierung von Lösungen werden vorgestellt.

In **Kapitel 3** wird eine formale Abstraktion von Spielen und ihrer Suchbaumbasierten Lösung eingeführt, die zum theoretischen Verständnis beitragen und die Beschreibung der in dieser Arbeit vorgestellten Such- und Lernalgorithmen $UP-max^n$ und $TD-Prob^n(\lambda)$ erleichtern soll.

Kapitel 4 zeigt, welche Komponenten ein verallgemeinerter Suchalgorithmus aufweist. Die Auswirkung von unterschiedlicher Konfigurationen und Optimierungen wird besprochen. Ebenfalls wird gezeigt, wie der Suchalgorithmus

ausgehend von zwei-Spiele-Spielen (2S-Spiele) erweitert werden kann, um für ein breiteres Spektrum an Spielen optimale Ergebnisse zu liefern.

Anschließend wird in **Kapitel 5** der Suchalgorithmus $UPP-max^n$vorgestellt. Dieser realisiert den im vorherigen Kapitel vorgestellten allgemeinen Suchalgorithmus auch für Mehrspieler-Spiele (MS-Spiele). Dabei wird versucht, mittels der Entropie-Nutzenfunktion $U_{H'}$ das MS-Entscheidungsproblem zu lösen.

Die neu entwickelte selbstlernende Methode $TD-Prob^n(\lambda)$ wird in **Kapitel 6** erklärt. Sie erweitert das Konzept der temporalen Differenz zur Optimierung von Gewichtsvektoren einer Bewertungsfunktion, um somit auch den Einsatz in nicht-deterministischen MS-Spielen zu ermöglichen.

Die Ergebnisse der hier vorgestellten Algorithmen werden in **Kapitel 7** besprochen. Sie wurden innerhalb des KI-Frameworks *jGameAI* für mehrere Spiele erstellt. Zusätzlich werden einige zukünftige Forschungsarbeiten angeführt, die sich an diese Diplomarbeit anknüpfen lassen.

Das KI-Framework jGameAI wird im **Kapitel 8** vorgestellt. Die Projektstruktur und die zugrunde liegenden Implementierungskonzepte werden erläutert. Dazu ist eine Übersicht von allen bisherigen Spieleentwicklungen und aktuellen Projekten aufgeführt.

Der *Java*-Quellcode der beiden vorgestellten Algorithmen $UPP-max^n$ und $TD-Prob^n(\lambda)$ ist unter dem **Kapitel 9** (APPENDIX) zu finden. Dazu eine Beschreibung der erfundenen Spiele 3-gewinnt! und 3-gewinnt!*.

Vielen Dank an Prof. Dr. Raúl Rojas und Marco Block-Berlitz, die meine Diplomarbeit betreut und mir dieses Thema ermöglicht haben.

Spiele haben mich auch nach meiner Kindheit fasziniert. Sie bieten die friedliche Möglichkeit, Lösungskonzepte gegeneinander antreten zu lassen und den Geist unterhaltsam anzuregen. Die Erfahrungen, die man bei der Lösungssuche in unterschiedlichen Spielsituationen macht, bereichern oft auch andere Lebenssituationen.

Eigentlich unterstütze ich es nicht unbedingt, dass immer mehr die von und für Menschen erdachten Gesellschaftsspiele von Computern gelöst werden. Die Kenntnis der spieltheoretischen Lösung eines Spiels beschränkt auf gewisser Art unser Bestreben uns ständig weiter zu verbessern, da sie das offensichtliche Ende dieser Entwicklung vorführt. Eine Forschung darüber hinaus macht wenig Sinn.

Genarrt fühlen wir uns zudem vor allem dadurch, dass die Computerspieler wenig vom Spiel erfasst haben, sondern einfach nur konzentrierter und schneller in die Zukunft rechnen können. Dabei erfreuen sich doch bereits auch schwächere menschliche Spieler an der Schönheit eines Spiels und eigener Lösungswege, selbst wenn diese zur Niederlage führten.

Lernfähige Konzepte für Computerspieler versuchen im Gegensatz dazu, etwas mehr von der menschlichen Herangehensweise zu verstehen und erfolgreich einzusetzen. Von diesem Prozess profitieren auch wir Menschen: wir müssen das eigene Denken genauer analysieren und erfahren somit mehr über uns selbst.

Es macht mir daher große Freude, mich im Rahmen der AG-Spieleprogrammierung an der Freien Universität Berlin damit beschäftigen zu können. Ich bedanke mich für die Unterstützung bei der Entwicklung von *jGameAI* durch Marco Block-Berlitz, Miao Wang, Johannes Kulick und Benjamin Bortfeldt.

Hiermit bestätige ich, dass ich diese Diplomarbeit und alle Abbildung (bis auf die Abbildungen 2.1 und 8.14) selbstständig angefertigt und alle verwendeten Textpassagen, Formeln und Zitate von Dritten als solche markiert und im Quellenverzeichnis angegeben habe.

Berlin, den 02.12.2008 ___________________

Maro Bader

Kapitel 2

Verwandte Arbeiten

Zur Entwicklung eines allgemein gültigen Lösungsansatzes für Spiele müssen die unterschiedlichen Herausforderungen der Spiele erkannt werden. Diese resultieren oft aus der unterschiedlichen Komplexität von Spielen und ihren möglichen Lösungsansätzen. Welche Art von Spielen gibt es und wie kann man die Komplexität von diesen bestimmen?

Die Möglichkeiten und Grenzen von Suchbaumbasierten Verfahren zur Findung optimaler Strategien werden in der Literatur ausführlich dieskutiert. Die gefundenen Erkenntnisse dienten als Grundlage für den allgemeinen Suchalgorithmus $UPP - max^n$.

Der Einsatz von selbstlernenden Methoden wird in einigen Spielen bereits unternommen und muss bei der Konzeption eines verallgemeinerten Lernverfahrens untersucht werden. Dazu werden für die Beschreibung der Funktionsweise des Lernverfahrens $TD - Prob^n(\lambda)$ vor allem maschinelle Lernmethoden mittels temporaler Differenz näher betrachtet.

2.1 Komplexität von Spielen

Die häufig in der gegenwärtigen Literatur untersuchten Spiele reichen von komplexen 2S-Spielen mit perfekter Information, wie etwa GO [Schraudolph 1994, Wang/Y 2007], bis hin zu MS-Spielen mit Zufallsereignissen, Mehrfachzügen, unvollständiger Information und teilweise unbekanntem Ausgang, wie etwa Poker [Billings 2006, Schauenberg 2006], Bridge [Ginsberg 2001], Hearts, Spades [Sturtevant 2006-3, Sturtevant 2002] und Siedler von Catan [Thomas 2002].

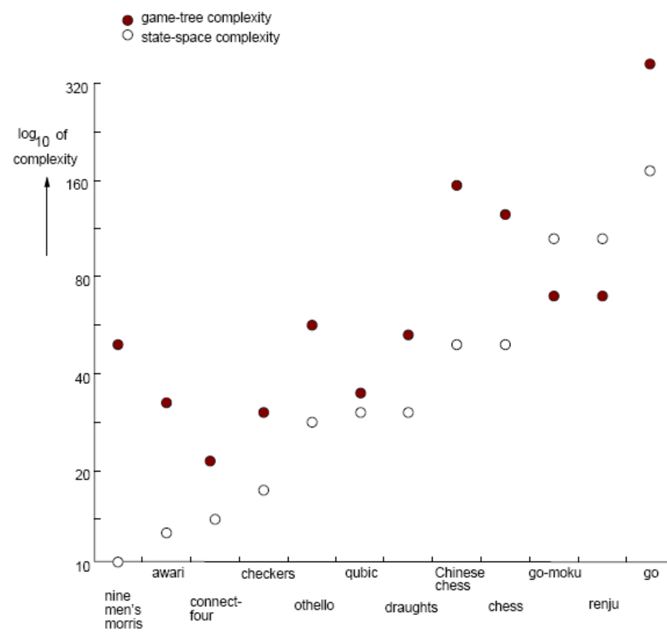


Abbildung 2.1: Übersicht über die Zustandsraumgröße und Spielbaumkomplexität für gängige Spiele. (entnommen aus [Allis 1994])

Die unterschiedlichen Eigenschaften dieser Spielelemente eignen sich oft nicht als Vergleichsmaß für jeweilige Lösungsmöglichkeiten. Dieses kann sinnvoller anhand der **Zustandsraumgröße** (*state-space complexity*) und der **Spielbaum-Komplexität** (*game-tree complexity*) dargelegt werden. Erstere beschreibt, oft nur als obere Schranke oder Annäherung, die Anzahl der legal erreichbaren Stellungen ab Spielbeginn. Die Spielbaum-Komplexität gleicht der Anzahl an Blattknoten des Suchbaumes, der beginnend bei der Startstellung alle möglichen Partien enthält.

Als konvergente Spiele werden solche bezeichnet, deren Zuglisten im Verlauf einer Partie kleiner werden. Sie besitzen häufig die Möglichkeit des Einsatzes von Endspieldatenbanken, die von bekannten terminalen Stellungen aus rückwärts gehend Spielstellungen ermitteln, die zu diesen führen. (vgl. beispielsweise GO und Schach in [Allis 1994]).

2.2 Suchbaumbasierte Verfahren zur Lösung von Spielen

Die grundlegende Herangehensweise, starke Computerspieler mittels Suchbaumansätzen zu konzipieren, hat sich seit Shannons Arbeit über Schachprogramme zu Beginn der 50er Jahre [Shannon 1950] kaum verändert. Shannon schlug darin vor, einen Suchbaum zukünftig erreichbarer Spielstellungen aufzubauen und mittels des Minimax-Algorithmus nach dem von beiden Spieler am wahrscheinlichsten gewählten Spielverlauf zu suchen. Da im Schach wie auch in den meisten noch nicht gelösten Spielen das Aufbauen des kompletten Suchbaumes (noch?) nicht möglich ist [Allis 1994], schlug Shannon das vorzeitige Abbrechen der Suche und eine Bewertung der bis dahin im Suchbaum erreichten Blattknoten mittels einer heuristischen Bewertungsfunktion vor (vgl [Russel/Norvig 2004]).

Die Suche bleibt dabei bis auf einige Optimierungsschritte und Erweiterungen vom Prinzip her für die meisten 2S-Spiele gleich. Breite Verwendung findet etwa die Alpha-Beta-Optimierung, die für eine optimale Strategie irrelevante Teilbäume während der Suche abschneidet und diese somit beschleunigt [Knuth 1975, Schaeffer 1989].

Dieser Ansatz bewährte sich und konnte im späteren Verlauf durch Hinzunahme einer Nutzenfunktion [Luce 1957] und einer Erweiterung des Suchalgorithmus auch für MS-Spiele genutzt werden [Luckhardt 1986]. Wenig später wurde auch die Alpha-Beta-Erweiterung für MS-Spiele weiterentwickelt [Korf 1991].

Aktuelle Forschungen bemühen sich um weitere Optimierungsmöglichkeiten, welche die gesteigerte Komplexität von MS-Suchbäumen bewältigen können (z.B. [Sturtevant 2004, Peterson 2002]).

In Alternative zur suchbaumbasierten Lösung von Spielen, also die reine Rechenkraft (*brute force*), werden häufig mathematische Beweise, Monte-Carlo-Suchmethoden (z.B. [Wang/Y 2007]), Eröffnungsgbücher und Endspieldatenbanken [Schaeffer 2007] sowie wissensbasierte Verfahren (z.B. [Yoshizoe 2007]) verwendet. Eine gute Übersicht aller eingesetzten Methoden zur Lösung von Spielen bietet [Allis 1994]. Oft ist eine Kombination mehrerer Ansätze notwendig, um die Komplexität eines Spiels wie beispielsweise Dame zu bewältigen [Schaeffer 1996, Schaeffer 2007].

2.3 Klassifizierung von Lösungen

Ein natürliches Bestreben von Menschen ist es, in den von ihnen gespielten Spielen sich konstant zu verbessern und möglichst unschlagbar zu werden. Die Grenze dieses Bestrebens ist in der spieltheoretischen Lösung eines Spieles erreicht. Perfekte Spieler, die auf diese zurückgreifen können, kennen den theoretischen Ausgang eines Spiels (Sieg, Niederlage oder Unentschieden aus Sicht der beginnenden Spieler) und sind in der Lage, optimale Siegstrategien zu entwickeln. Es werden dafür drei Lösungsklassen unterschieden [Allis 1994]:

- Die ultraschwache Lösung (*ultraweakly solved*) benennt nur den Ausgang eines Spiels vom ersten Zug an, wenn sich alle Spieler fehlerlos verhalten, ohne die Strategie selbst zu benennen. Eine solche Aussage liegt zum Beispiel für das von John F. Nash miterfundene Spiel *Hex* vor [Rijswijck 2003]: der zuerst ziehende Spieler kann stets gewinnen, wenn er perfekt spielt. Die dafür benötigte Strategie bleibt aber für Varianten von *Hex* mit größeren Spielbrettern ($> 9 \times 9$) bis heute noch unbekannt [Rijswijck 2003, Yang 2002]. Bei der spieltheoretischen Aussage zu einem Spiel muss nicht immer zwangsläufig der erste Spieler gewinnen. Bei der 6×6-Variante des Brettspiels *Othello* konnte bewiesen werden, dass der zweite Spieler in adäquater Reaktion auf alle vorgehenden Züge einen Sieg erzwingen kann [Feinstein 1993, Herik 2002].

- Die schwache Lösung (*weakly solved*) umfasst solche Lösungen, die sowohl den spieltheoretischen Ausgang eines Spieles, als auch die von der Startstellung ausgehende perfekte Strategie benennen können. In [Allis 1994]

wird beispielsweise die schwache Lösung des Spiels *Go Moku* beschrieben, einer einfacheren Variante des japanischen Brettspiels *GO*. Es lässt sich zeigen, dass der erste Spieler einen Sieg erzwingen kann. In Europa eventuell bekannter ist das ebenfalls schwach gelöste Spiel *4-Gewinnt*! [Allen 1989, Allis 1988]. Auch hier gewinnt der zu Beginn ziehende Spieler.

- Die höchste Stufe der Lösung ist die starke Lösung (*strongly solved*). Dabei liegt nebem dem spieltheoretischen Ausgang eines Spiels auch die perfekte Strategie vor, ausgehend von jeder beliebigen erreichbaren Stellung im Spiel. Eine solche Lösung kann für jede Partei und Stellung definieren, welcher Spielausgang noch maximal zu erreichen ist und wie der Weg dorthin verläuft. Stark gelöste Spiele wie *Awari* [Romein 2002], können somit nicht mehr von Menschen überlistet werden.

Ist ein Spiel einmal gelöst oder ein schwer zu schlagender Computerspieler geschaffen, so verliert die Lösung des Spiels schnell an Reiz für die Forschung. Wie lange die Motivation, ein Spiel zu lösen dabei anhalten kann, zeigt das Beispiel des Brettspiels *Dame* (*checkers*).

Bereits Mitte der 50er Jahre wurde von Samuel ein Dameprogramm entwickelt [Samuel 1959], welches nach dem ersten Sieg über einen Damemeister 1962 von vielen Zeitgenossen als ein Meilenstein der künstlichen Intelligenz betrachtet wurde. Es handelte sich dabei um einen der ersten selbstlernenden Computerspieler, der den Grundstein für viele Weiterentwicklungen legte.

Es dauerte allerdings ganze 45 Jahre, bis *Dame* wirklich als gelöst betrachtet werden konnte [Schaeffer 2007]. Die Arbeitsgruppe rund um Jonathan Schaeffer schaffte es nach knapp 20 Jahren Computer-unterstützer Berechnung, das Spiel *Dame* schwach zu lösen und bestätigte mit der Erkenntnis, dass das Spiel bei perfektem Spiel beider Seiten unentschieden endet, die Vermutung vieler professioneller Damespieler [Schaeffer 2007].

2.4 Einsatz von Lernmethoden

Der Einsatz von selbstlernenden Methoden mit Hilfe temporaler Differenz wurde seit der Erstveröffentlichung durch Sutton [Sutton 1990] in vielen Spielen untersucht und erfolgreich angewandt (unter anderem [Baxter 1998, Block 2004, Ghory 2004, Schraudolph 1994, Tesauro 1995]). Alle Ansätze basieren auf der gleichen Verknüpfung zweier grundlegender algorithmischer Konzepte: Monte-

Carlo-Verfahren und dynamische Programmierung [Sutton 1998]. Anhand dieser maschinellen Lernmethode lassen sich die von Menschen initial gesetzten Gewichte von Stellungsbewertungen optimal an ein Spiel anpassen und somit die Spielstärke erhöhen.

Das in [Block 2004] angewandte Verfahren $TD - Leaf - ComplexEval(\lambda)$ lieferte für Schach sehr viel versprechende Ergebnisse. Eine deutliche Steigerung der Spielstärke des Schachprogrammes $FUSc\#$ [Block 2005] konnte nach einem Trainingszyklus von nur 72 erlernten Partien festgestellt werden.

Eine gute Übersicht selbstlernender Computerspieler für eine breites Spektrum an Spielen und die verwendeten Methoden findet sich in [Ghory 2004].

Weitere Lernmethoden werden bei MS-Spielen vor allem bei der Modellierung der Gegner und dem Lernen ihrer Verhaltensweisen eingesetzt [Thomas 2003, Schauenberg 2006].

Kapitel 3

Abstraktion von Spielen

3.1 Spiele als nicht deterministische Zustandsautomaten

Ein Spiel (*game*) G wird beschrieben als eine Tupel $(\mathcal{S}, \delta, \mathcal{P}, \upsilon)$. Die Menge $\mathcal{S}$ (*states*) entspricht allen legalen Spielzuständen. Mit der Zustandsübergangsfunktion δ kann man mittels eines legalen Spielzuges (*action*) a von einem Zustand S_t ausgehend in den Folgezustand S_{t+1} gelangen. Alternativ zu der Notation

$$\delta(S_t, a) = S_{t+1}$$

.

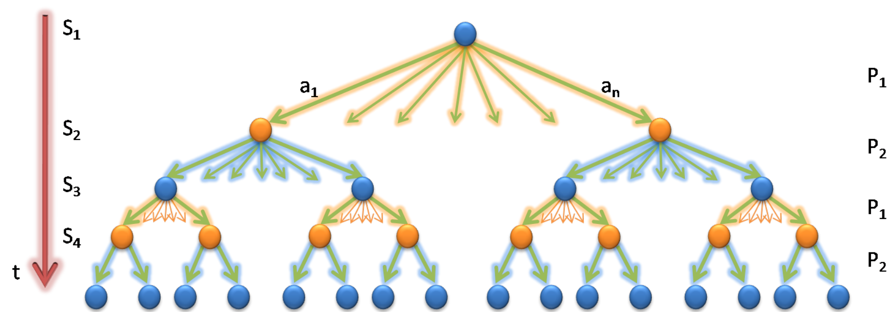


Abbildung 3.1: Graphische (partielle) Darstellung eines abstrakten Spiels mit zwei Spielern.

kann man auch die Schreibweise

$$S_{t+1} = S_t \circ a$$

verwenden. Ein Zustand $S_t \in \mathcal{S}$ wird nach dem Ausführen von genau t Spielzügen erreicht (vgl. Notation aus Kapitel 10.2. in [Ertel 2008]). Diese Notation wurde gegenüber anderen etwa aus [Russel/Norvig 2004, Rieck 2007] bevorzugt, da sie sich für die später eingeführten maschinellen Lernmethoden aus Kapitel 6 besser eignet.

In der Menge $\mathcal{P}$ sind die am Spiel beteiligten Spieler (*players*) enthalten. Die Spielstandsfunktion v liefert zum Zustand den aktuellen Spielstand für jeden Spieler (Sieg, Niederlage und Unentschieden).

Zur Anschauung hilft es, Spiele als gewichtete, gerichtete Graphen darzustellen. Dabei werden Zustände als Knoten und Spielzüge als gerichtete Kanten interpretiert. Die Übergänge von einem Zustandsknoten zu seinen jeweiligen Folgeknoten entsprechen dabei den mit δ beschriebenen Zustandsübergängen (vgl. [Russel/Norvig 2004]).

Für jeden Zustand S_t liegt eine Menge $\mathcal{A}_t$ von legalen Spielzügen vor. Für $\forall a_i \in \mathcal{A}_t$ führt a_i über $\delta(S_t, a_i)$ in einen neuen, legalen Spielzustand S_{t+1}. Die Anzahl der Folgezustände von S_t ist also gleich der Anzahl der Elemente in $\mathcal{A}_t$, welche von den Spielregeln und der aktuellen Spielsituation abhängig sind. $\mathcal{A}_t$ entspricht der Menge aller ausgehenden Kanten des Zustandsknoten für S_t in G.

In einem Spielzug a_i festgehalten ist der Index j des Spielers $(a_{i \rightarrow P_j})$, der diesen Zug ausführen kann. In der graphischen Darstellung wird die entsprechende Kante mit einer eindeutigen Spielerfarbe gekennzeichnet. In den meisten rundenbasierten Spielen enthält die Menge $\mathcal{A}_t$ zu keinem Zeitpunkt Spielzüge mit unterschiedlichen Spielerindizes. In Echtzeitspielen können zu einem Zeitpunkt auch mehrere Spieler an der Reihe sein, üblicherweise dann alle. Dies bietet sich häufig bei Spielen an, die in einer simulierten Umwelt agieren können und dabei kein festes Protokoll einhalten müssen.

3.1.1 Terminale Zustände

Sollte $\mathcal{A}_t$ für einen Zustand S_t die leere Menge sein, so ist S_t ein terminaler Zustand und die Partie kann von hier aus nicht weiter gespielt werden. In $\mathcal{S}$ befinden sich der Startzustand S_0 des Spiels (gelber Knoten), sowie alle Zwischen-

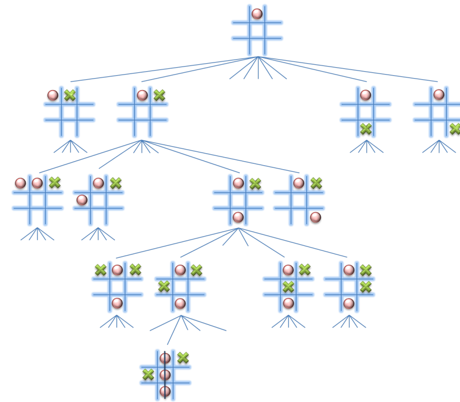


Abbildung 3.2: Graphische partielle Darstellung eines TicTacToe-Spiels.

und terminalen Zustände (roter Schatten). Nicht näher betrachtete Teilbäume können in der graphischen Darstellung durch ein graues Dreieck ersetzt werden. Sie sind dann "nicht expandiert".

Die Menge $\mathcal{S}$ muss nicht zwingend vollständig bekannt sein. Es reicht, wenn für jeden legal erreichbaren Zustand S_t eine Funktion (Zuggenerator) vorliegt, welche die Menge $\mathcal{A}_t$ erzeugt. Eine Partie ω^n bezeichnet den Pfad von einem Startzustand über n gespielte Spielzüge und zwischenzeitlich erreichte Spielzustände $\omega_1^n, \omega_2^n, \ldots, \omega_n^n$ zu einem Endzustand.

Für alle Zustände S_t liefert die Siegerfunktion (*victory*) $v(S_t)$ den aktuellen Spielstand als Vektor $\overrightarrow{v_t}$ zurück. Für jede Komponente v_{t_i} des Vektors gilt:

$$v_{t_i} = \begin{array}{ll} +eval_{max} & : P_i \quad gewonnen \\ -eval_{max} & : P_i \quad verloren \\ 0 & sonst \end{array}$$

(wobei $eval_{max} > 0$ ist).

Üblicherweise wird $\overrightarrow{v_t}$ nur dann nicht der Nullvektor sein, wenn es sich bei S_t um ein terminalen Zustand handelt. Sollte dies auch bei einem terminalen Zustand der Fall sein, so gilt die Partie als unentschieden. In MS-Spielen mit n Spielern kann es mehrere Sieger und Verlierer gleichzeitig geben. Somit liegt die Summe aller Komponenten von $\overrightarrow{v}$ im Intervall $[-n \cdot eval_{max},\ n \cdot eval_{max}]$ und kann dabei als Wert nur ganzzahlige Vielfache von $eval_{max}$ annehmen.

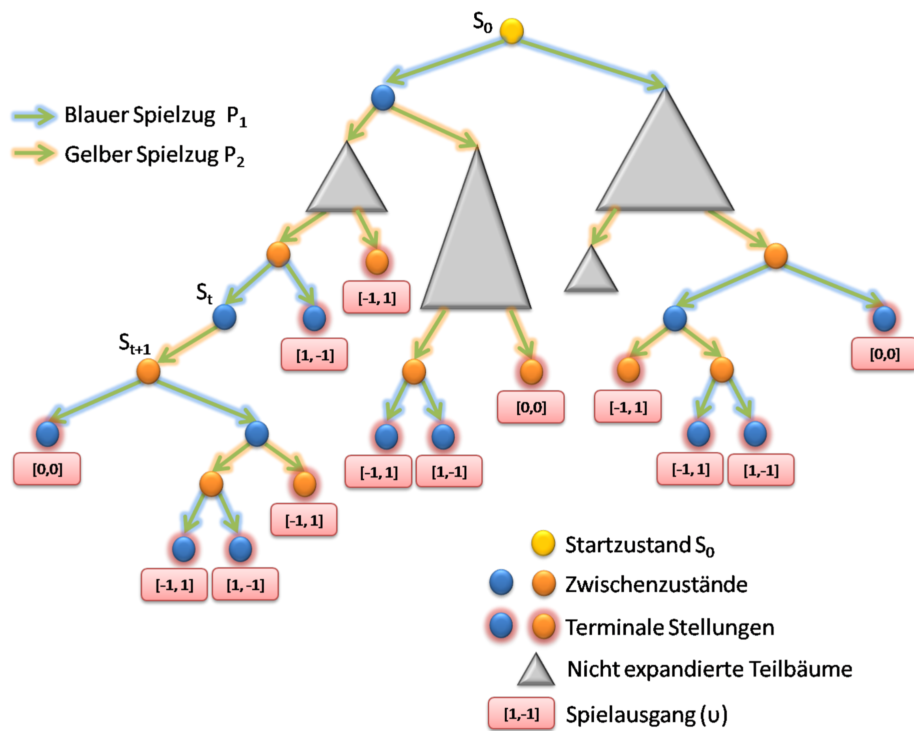


Abbildung 3.3: Komplexer Spielgraph mit Teilbäumen, gekennzeichneten terminalen Zuständen und Spielausgängen.

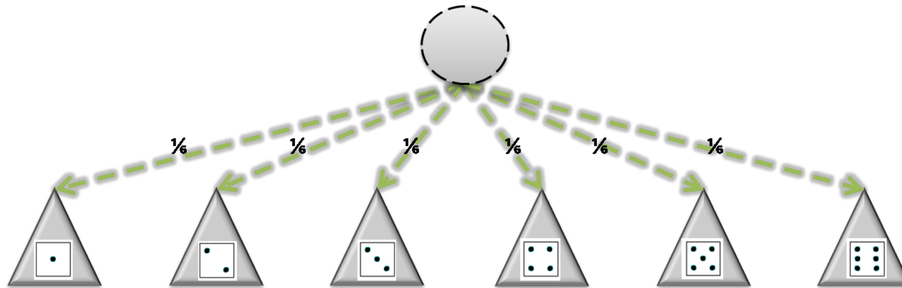


Abbildung 3.4: Zufallsknoten und -kanten werden grau und unterbrochen gezeichnet. Hier am Beispiel eines Würfelwurfes.

3.1.2 Zufallsereignisse und unvollständige Information

In Spielen mit Wahrscheinlichkeitselementen wie etwa das Würfeln oder die Verteilung von Karten gibt es Zustände, die von alleine über ein Zufallsereignis a^* mit einer bekannten Wahrscheinlichkeit p_{a^*} in einen Folgezustand $S_t \circ a^*$ übergehen. Es handelt sich dabei um nicht-deterministische Zustandsübergänge, sobald mehr als ein Zufallsereignis von einem Zustand aus möglich ist.

Zufallsereignisse können nicht von Spielern erzwungen werden. Ein Zustand ist entweder ein Zufallszustand, falls Zufallsereignisse von diesem ausgehen, oder ein Spielzustand, falls deterministische Spielerzüge möglich sind.

Ein Zustand kann nur Zufalls- oder Spielzustand sein. Für einen Zufallszustand gilt:

$$\sum_{i}^{A_t} p_{a_i^*} = 1 \tag{3.1}$$

.

Die Knoten der nicht-deterministischen Zustände und die Kanten der Zufallsereignisse werden in der Graphendarstellung unterbrochen gezeichnet, die Wahrscheinlichkeit der Zufallsereignisse als Kantengewicht angeführt.

In Spielen mit perfekter Information liegen allen Spielern zu jedem Zeitpunkt einer Partie die gesamten Informationen über den aktuellen Zustand vor. Man spricht von imperfekter Information, wenn dies nicht der Fall ist (vgl. *perfect/imperfect information* in [Kuhn 1953, Rieck 2007]). In die Zukunft blickend, werden unterschiedliche Spieler dann unterschiedliche zukünftige Zustände ermitteln, da die Menge $\mathcal{A}_t$ von der verfügbaren Information über den Zustand S_t abhängt.

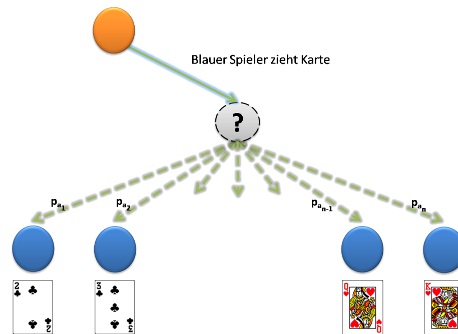


Abbildung 3.5: Ein aus Sicht des gelben Spielers verdeckter Zug: Blau zieht eine Karte.

Eine noch feinere Unterteilung wird bei Spielen zwischen vollständiger und unvollständiger Information vorgenommen [Rieck 2007]. Auch in Spielen mit perfekter Information können Spieler private Informationen besitzen, die sie anderen vorenthalten. Dies ist zum Beispiel beim Spiel *Doppelkopf* der Fall, in dem zwar alle Stiche offen gespielt werden, zu Beginn einer Partie aber noch nicht jeder seinen möglichen Mitspieler kennt. Im Verlauf einer Partie offenbaren sich die Parteien den Spieler zu unterschiedlichen Zeitpunkten. Erst ab diesem Moment ist die Information vollständig.

Kann ein Spieler über die möglichen Züge zu einem Zustand nur Vermutungen anstellen, so nennt man diese Züge verdeckte Züge. Sie besitzen ähnlich wie bei Zufallsereignissen eine bekannte Wahrscheinlichkeit, die sich aus der Menge aller in Frage kommenden Züge ableiten lässt.

Ein Beispiel für einen verdeckten Zug ist das verdeckte Ziehen einer Karte vom Stapel. Aus der Menge der noch nicht gesichteten Karten kann jeder Spieler eine eigene Menge $\mathcal{A}_t$ von Zufallsereignissen samt ihren Wahrscheinlichkeiten ermitteln. Es gibt somit vom Informationsstand abhängige, unterschiedliche Blickweisen (*views*) auf das Spiel (vgl. [Russel/Norvig 2004]).

Für Knoten, an denen verdeckte Züge möglich sind, gelten die gleichen Bedingungen wie für Zufallsknoten: Normierung der Wahrscheinlichkeiten zur Summe 1 und Nicht-Determinismus. Zur besseren Darstellung werden auch diese grau und unterbrochen gezeichnet. In Abschnitt 7.2.4 werden die Probleme und deren Lösungen aufgezeigt, die durch verdeckte Züge auftreten können.

3.1.3 Spielbaum und Partien

Im Folgendem bezeichnet S_t sowohl den Zustand als auch den Knoten, der jenen in einem graphischen Kontext beschreibt. Analoges gilt für Zustandsübergänge und die Kanten. In dieser Notation von Spielen ist es ausgeschlossen, dass Zustandsknoten mehr als eine eingehende Kante besitzen. Somit handelt es sich bei dem ein Spiel beschreibenden gerichteten Graphen eigentlich um einen Baum mit dem Startknoten als Wurzel und allen Endknoten als Blätter.

Dieser "Spielbaum" wird der Einfachheit halber ebenfalls mit G angeführt. Die Menge der Pfade von S_0 zu allen Blattknoten und somit allen terminalen Stellungen innerhalb von G entspricht der Menge aller spielbaren Partien Ω.

In Ω sind keine verdeckten Züge enthalten, da hier nur die real spielbaren Partien enthalten sind. Ω ist je nach Spiel und Spielregeln endlich (*Stein-Schere-Papier, TicTacToe, Schach, 4 gewinnt...*), abzählbar (*Dame, Poker, Siedler von Catan,*) oder überabzählbar (Simulationen, Spiele mit kontinuierlichem Handlungsraum, Asteroids). Ein Wort ω aus Ω entspricht einer Liste von nicht verdeckten Spielzügen und eingetretenen Zufallsereignissen.

3.2 Suchbaum und Strategie

Ziel der Suche ist es, für den ziehenden Spieler im Hinblick auf jede Spielsituation den stärksten Zug zu ermitteln. Was dabei das Vergleichsmaß ist, wird in Abschnitt 3.3 erläutert. Um die Auswirkungen der zur Verfügung stehenden Züge zu vergleichen, muss mindestens ein Zug in die Zukunft geblickt werden. Dieser Zukunftsraum beginnt am aktuellen Zustandsknoten und geht über alle möglichen nächsten Spielzüge oder Zufallsereignisse zu Folgezuständen. Für eine solche Suche eignet sich das Betrachten des Spielbaumes G.

3.2.1 Suchbaum

Als Suchbaum B_t^d bezeichnet man den Teilbaum eines Spieles, der ausgehend von einem Zustand S_t alle in der Zukunft über maximal d Spielzüge erreichbaren Folgezustände beinhaltet. S_t ist dabei ein im Verlauf einer Partie tatsächlich erreichter Zustand. Die Blattknoten von B_t^d entsprechen entweder terminalen Stellungen, die nicht mehr in Folgezustände überführbar sind, oder Zustandsknoten, deren Kantenentfernung zur S_t gleich d ist.

Für jedes Spiel und jeden Spieler existiert theoretisch ein maximaler Such-

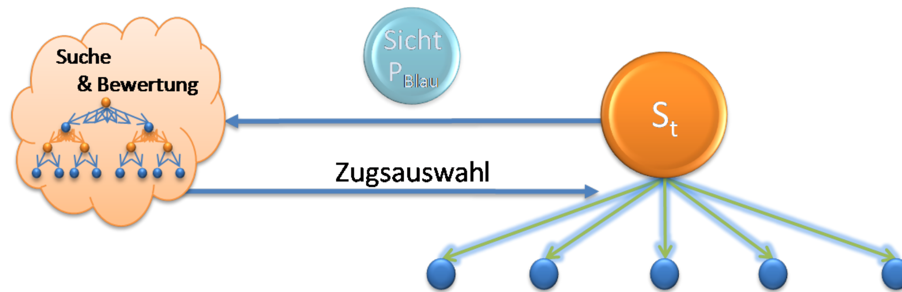


Abbildung 3.6: Die Suche von Spieler 3 nach dem nächsten Zug bleibt nach aussen hin geheim und verwertet nur die eigene Sicht des Zustands.

baum B_0^∞ , der beginnend vom Startzustand S_0 des Spiels alle möglichen Partien und somit alle möglichen Strategien enthält. Suchbäume werden aus der Sicht eines Spielers erstellt und können daher je nach Informationslage auch verdeckte Züge enthalten. Der Spielbaum G ist in B_0^∞ enthalten ($G \subseteq B_0^\infty$), da in den verdeckten Zügen auch der real mögliche enthalten ist. In Spielen mit vollständiger Information gilt sogar $G \equiv B_0^\infty$.

3.2.2 Strategie

Ein Pfad in B_t^d von der Wurzel S_t zu einem Knoten S_{t+j} bestehend aus j legalen Spielzügen und Zufallsereignisse mit $j \leq d$ wird Strategie Π_t^j für den Zustand S_t genannt. Eine Strategie ist ein Teilabschnitt mindestens eines Wurzelpfades aus B_0^∞, der über die Knoten S_t und S_{t+j} verläuft. Die Züge und Zufallsereignisse der Strategie Π_t^j können analog wie bei einer Partie ω über direkte Referenzierung

$$\pi_t^j[1],\ \pi_t^j[2], \ldots,\ \pi_t^j[j]$$

adressiert werden (vgl. Kapitel 10.2. in [Ertel 2008]).

Strategien sind ein wichtiges Werkzeug der suchbaumbasierten Suche nach besten Zügen. Wurde die “sinnvollste” Strategie von einem Zustand S_t ausgehend ermittelt, so entspricht $\pi_t^j[1]$ dem aktuell besten Zug aus der Sicht des suchenden Spielers. Damit der erste Zug eines Spielers ein spielbarer ist, wird die Suche überlicherweise erst dann gestartet, wenn der suchende Spieler mit einem eigenen deterministischen Zug an der Reihe ist.

Christian Rieck verwendet in seinem Buch eine leicht abweichende Notation für Strategien ([Rieck 2007], Kapitel 4). Dort bezeichnet eine Strategie S

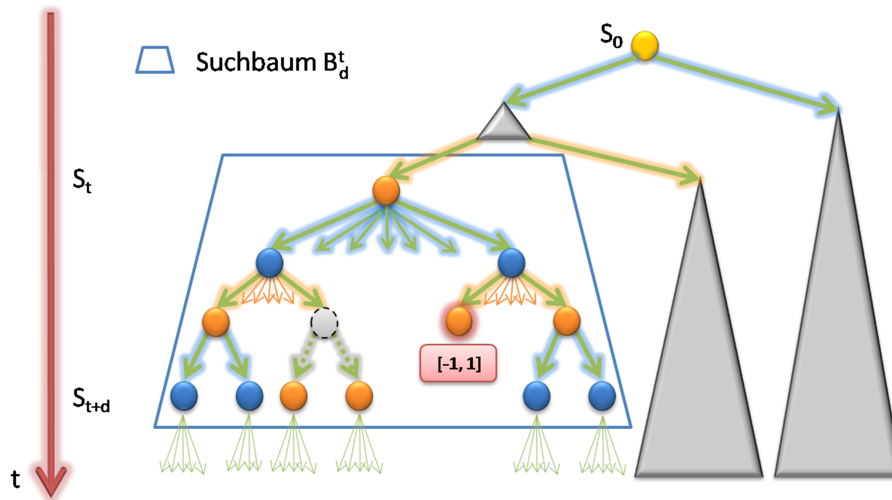


Abbildung 3.7: Ein Suchbaum B_t^d aus der Sicht des blauen Spielers als Teilbaum des maximalen Suchbaumes B_0^∞.

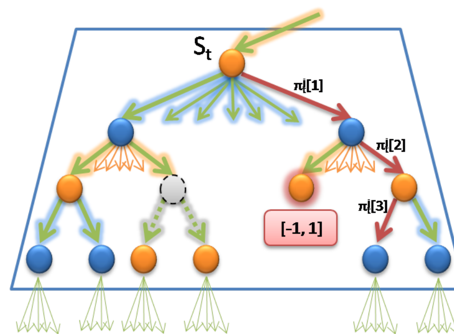


Abbildung 3.8: Rot gekennzeichnet: Eine Strategie $\Pi_t^3 = (\pi_t^3[1], \pi_t^3[2], (\pi_t^3[3])$ im Suchbaum B_t^3 .

"einen vollständigen Verhaltensplan" für alle Stellungen eines Spiels. Die hier eingeführte Strategie Π_t^j wäre in dem Fall nur ein Teilwort der Länge j eines Strategievektors $\hat{s} = (\hat{s}_1, \hat{s}_2, ..., \hat{s}_{\bar{I}})$ im Rieck'schen Sinne.

Da aber gerade dieser vollständige Verhaltensplan für Spiele Ziel der Erforschung der Künstlichen Intelligenz ist, kann der Rieck'sche Strategiebegriff praktisch nur für gelöste Spiele in Frage kommen. Für die im späteren Verlauf untersuchten Spiele eignet sich daher der Begriff einer auf einen beschränkten Suchraum basierenden Strategie besser und entspricht auch eher einer technischen Realisierung, wie sie beispielsweise im Framework *jGameAI* vorgenommen wird. In Abschnitt 3.2.3 wird die Erweiterung einer linearen Strategie zu einem Strategiebaum beschrieben, um so einen Verhaltensplan für alle im Suchbaum entlang eines Wurzelpfades auftretenden Zufallsereignisse zu beschreiben.

3.2.2.1 Spekulative Strategien

Eine Strategie heißt spekulativ, wenn sie mindestens ein Zufallsereignis oder einen verdeckten Zug enthält. Die Wahrscheinlichkeit, dass eine spekulative Strategie eintritt, entspricht dem Produkt aller Wahrscheinlichkeiten der Zufallsereignisse und verdeckten Züge entlang der Strategie

$$p(\Pi_t^j) = \prod_{i=1}^{j} \pi_t^j[i]$$

.

Ein Suchbaum, welcher keine spekulativen Strategien enthält, ist ein deterministischer Suchbaum, da jede Strategie innerhalb des Suchbaumes von dem Spieler auch wirklich durchgespielt werden kann. In probabilistischen Suchbäumen existiert mindestens eine spekulative Strategie, welche selbst bei Absprache der Spieler zufallsbedingt eventuell nicht unbedingt gespielt werden kann, da sie zum Beispiel mindestens einen nicht möglichen verdeckten Zug enthält.

3.2.2.2 Siegreiche Strategien

Die von einer Strategie Π_t^j noch theoretisch erreichbaren terminalen Stellungen eines Spiels entsprechen allen Blattknoten des bei S_{t+j} beginnenden Teilbaumes vom Spielbaum G. Da der suchende Spieler eventuell mit verdeckten Zügen arbeiten muss, erweitert man diese Menge um die erreichbaren terminalen Stellungen des bei S_{t+j} beginnenden Teilbaumes innerhalb von B_0^∞.

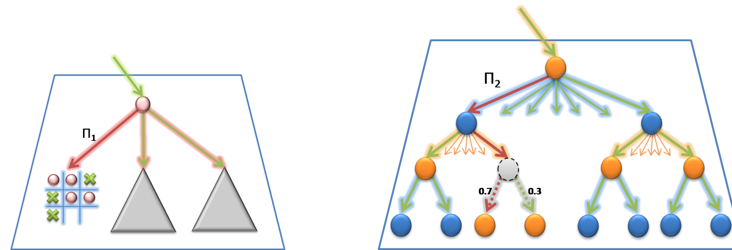


Abbildung 3.9: *Links*: Π_1 ist eine für Spieler X siegreiche Strategie, da alle theoretisch noch erreichbaren terminale Stellungen von Spieler X gewonnen werden. *Rechts*: Die Strategie Π_2 ist eine spekulative Strategie ($p(\Pi_2) = 1 * 1 * 0.7 = 0.7$), da sie über einen Zufallsknoten führt.

Eine Strategie Π_t^j heißt siegreich, wenn alle noch erreichbaren terminalen Spielstellungen zugunsten des suchenden Spielers einem Sieg entsprechen. Da B_0^∞ häufig nicht vorliegt, reicht für diese Aussage auch der Beweis, dass der bei S_{t+j} beginnende Teilbaum nur noch siegreiche Endzustände enthalten kann [Peterson 2002].

Befinden sich in einem Suchbaum nur siegreiche Strategien, so besitzt der suchende Spieler die Möglichkeit, den Sieg zu erzwingen, d.h. egal welche Antwortzüge die Gegner wählen, der suchende Spieler kann immer den Sieg herbeiführen. Man nennt dann S_t eine für den suchenden Spieler siegreiche Stellung. Der durch eine siegreiche Strategie erreichbare Zustand S_{t+j} ist somit zwangsläufig eine siegreiche Stellung für den Suchenden (vgl. Kapitel 4.7.3. in [Rieck 2007]).

3.2.3 Strategiebaum und Suchbreite

Bisher stellen Strategien einen Wurzelpfad innerhalb eines Suchbaumes dar. Eine solche Strategie nennt man reine Strategie, da sie zu jedem Zeitpunkt nur einen einzigen Zug als Antwort vorgibt. Sie kommt in Suchbäumen von deterministischen Spielen mit vollständiger Information vor.

In Spielen mit Zufallselementen oder unvollständiger Information kann es sein, dass eine reine Strategie selbst bei Absprache aller Spieler eventuell nicht durchgespielt werden kann, da Zufallsereignisse der Strategie nicht eintreten oder ein verdeckter Zug der Strategie nicht möglich war.

Um zu vermeiden, dass auch solche Strategien bei der Suche berücksichtigt werden, muss an Zufalls- und verdeckten Knoten eine Zerlegung der Strategie in Unterstrategien erfolgen, die alle jeweils mit der Wahrscheinlichkeit des Zufallsereignisses oder des verdeckten Zuges eintreten können. Aus dem Stra-

tegiepfad wird somit ein eigener Strategiebaum, der im weiteren ebenfalls als Strategie bezeichnet wird. Solche Strategien nennt man gemischte Strategien. Für gemischte Strategien liegt eine Wahrscheinlichkeitsverteilung aller in ihr enthaltenen reinen Strategien vor [Amann 1999].

Als Suchbreite oder Verzweigungsfaktor (*branching factor*) b wird die durchschnittliche Anzahl von Zustandsübergängen je Zustandsknoten eines Spiels bezeichnet [Ertel 2008]. Leider lässt sich b häufig nicht exakt für ein komplettes Spiel bestimmen, da es von Spielphase, Spielerverhalten und Zufallsereignissen abhängen kann oder die Anzahl aller möglichen Spielstellungen für eine genaue Berechnung zu groß ist.

Die Komplexität eines Spiels hängt entscheidend von b ab, da es die bedeutendste Auswirkung auf die Größe des Suchbaumes hat. Die Suchtiefe wird somit bei konstanter Suchzeit und Rechenleistung durch b beschränkt.

Die Notation $b(t)$ bezeichnet die Suchbreite an einem bestimmten Zustand S_t und ist damit gleich der Anzahl von Zustandsübergängen, die vom Zustand S_t aus wegführen. Bei konstanter Suchbreite lässt sich für einen Suchbaum B_t^d die Anzahl n der Gesamtknoten mit der Gleichung

$$n \leq \sum_{i=0}^{d} b^i = b^{d+1} - 1$$

berechnen. Nur wenn in B_t^d keine terminalen Zustände vorkommen, handelt es sich um einen vollständigen Suchbaum und für die Formel gilt das Gleichheitszeichen.

3.3 Bewertung und Suche

Eine vollständige Suche agiert auf B_0^∞ und versucht zu jedem Zeitpunkt, den Zug zu finden, der die Summe der Wahrscheinlichkeiten der noch nach diesem Zug spielbaren siegreichen Strategien maximiert. Dieses Vorgehen ist aufgrund der Komplexität von B_0^∞ und einer begrenzten Suchzeit häufig nicht möglich. Ein Verfahren, welches sich der vollständigen Suche mit annehmbarer Laufzeit annähert, bietet sich an.

Zwei Ansätze werden dafür gleichzeitig verfolgt:

- Reduzieren des Suchbaumes auf eine feste Höhe d, die Suchtiefe. Statt alle erreichbaren Zustände zu berechnen, berücksichtigt man nur noch solche,

die bis zu einer festen, vorgegebenen Tiefe vorkommen. Diese Tiefe kann man an die Zeitvorgaben, Spielphasen und die zur Verfügung stehende Rechenleistung anpassen. Innerhalb des so reduzierten Suchbaumes wird nach einer siegreichen Strategie der Länge d gesucht.

- In den meisten Fällen handelt es sich in diesem verkürzten Suchbaum bei Blattknoten nicht um terminale Zustände. Daher kann oft keine Aussage darüber getroffen werden, ob diese Stellung siegreich ist. Eine dem Spieler eigene Bewertungsfunktion $Eval$ untersucht diese Blattknoten und versucht, eine heuristische Aussage darüber zu treffen, ob die Zustände als eher siegreich oder als eher verloren gelten.

$Eval$ erweitert den diskreten Wertebereich $\{-eval_{max}, 0, +eval_{max}\}$von v zu einem kontinuierlichen Bereich und liefert auch für nicht-terminale Zustände sinnvolle Ergebnisse. Diese Lösung wurde ursprünglich von Shannon eingeführt und agiert auf dem in Abschnitt 3.2.1 beschriebenen höhenbeschränkten Suchbaum B_t^d [Shannon 1950].

Wie in Abbildung 2.1 leicht zu erkennen, liegt die Spielbaum-Komplexität für die dargestellten Spiele noch deutlich über der mit heutigen Rechnern zu bewältigenden Problemgröße. Gegenwärtig sind kaum Spiele mit einer Spielbaum-Komplexität von über 10^{12} gelöst worden [Heule 2007].

Somit lassen sich Suchbraumbasierte Lösungsverfahren niemals auf den gesamten Spielbaum anwenden und eine Beschränkung der Suchtiefe muss vorgenommen werden. Dies verdeutlicht die Wichtigkeit der Bewertungsfunktion, welche den Verlust der Weitsicht in die Zukunft bis zum Ende eines Spiels beheben muss.

Die Bewertungsheuristik muss eine vernünftige Prognose der nicht näher untersuchten Folgestellungen widerspiegeln. Gelingt dies nicht, scheitert damit auch die gesamte Suche, unabhängig von Rechenleistung und Optimierungsschritten des gewählten Suchbaumbasierten Algorithmus. Es lohnt sich daher, viel Mühe in die Bewertungsfunktion zu investieren, da sich hier die Intelligenz des Spielers und die Auswirkung auf die Qualität der Spiellösung am stärksten manifestieren.

3.3.1 Rationales Spielen durch Bewertungsfunktion

Jeder Spieler geht davon aus, dass seine Mitspieler rational vorgehen, und erstellt eine Prognose über ihr Verhalten und damit den Spielverlauf. Der Spieler

reagiert auf die zu erwarteten Spielzüge der Gegner mit dem eigenen stärksten Spielzug. Dabei ist der Begriff der Rationalität eine vom Spieler abhängige Einschätzung und spiegelt sich in der gewählten Suchmethode wieder. Unterschiedliche Bewertungen der gleichen Spielsituation führen zu einem daraus folgenden unterschiedlichen Spielerverhalten.

Die Funktion $Eval$ ist spielerabhängig und für die Gegner nicht einsehbar. Eine offene Bewertungsfunktion macht einen Spieler vorhersehbar. Für die Bewertung von Spielstellungen stehen nur die aus Sicht des bewertenden Spielers verfügbaren Informationen der Spielwelt zur Verfügung.

3.3.2 Ideale Bewertungsfunktion

Liegt eine ideale Bewertungsfunktion $Eval^*$ vor, so kann diese für jeden Zustand angeben, welche Spieler gewinnen und welche verlieren werden. Im Besitz von $Eval^*$ müsste keiner der Spieler tiefer als einen Zug in die Zukunft vorrausrechnen, um den besten Zug zu ermitteln. Das einfache Ausführen aller möglichen Spielzüge für den aktuellen Zustand und der Vergleich der daraus resultierenden Folgestellungen mit $Eval^*$ genügen in diesem Fall.

Im Spiel Schach würde dies zum Beispiel bedeuten, für jede mögliche Stellung direkt angeben zu können, ob und welcher Spieler einen Sieg forcieren kann. Das Spiel wäre damit stark gelöst [Allis 1994].

Die Funktion $Eval^*$ entspräche dann der Funktion v, die auch für nichtterminale Zustände diskrete Ergebnisse liefert, die ungleich dem Nullvektor sein können. Existiert $Eval^*$, so spricht man davon, dass ein Spiel *gelöst* ist. Ein menschlicher Spieler wäre in fairen Spielen gegen einen Computerspieler, der seine Entscheidungen mit Hilfe von $Eval^*$ trifft, zu keinem Zeitpunkt in der Lage, diesen zu schlagen.

In vielen Spielen ist es fraglich, ob $Eval^*$ überhaupt jemals gefunden werden kann. Ein Weg, dies zu erreichen besteht darin, B_0^∞ aufzuspannen und jeden Spielverlauf zu analysieren. Handelt es sich um ein entschiedenes Spiel, so speichert man für jeden zwischenzeitlich erreichten Zustand in einer Tabelle, welche Spieler am Ende des Spielverlaufs noch gewinnen können und wie der zu spielende Verlauf hin zu der siegreichen Stellung aussieht. Auf diese Weise wurde zum Beispiel das Spiel *Awari* stark gelöst [Romein 2002]. Eine gute Übersicht über weitere erfolgreich eingesetzte Methoden zur starken spieltheoretischen Lösung von Spielen bietet [Heule 2007] an.

Der Ansatz über eine vollständige Zustandstabelle würde bedeuten, den

Spiel	# Spielstellungen	Gelöst?	Konvergent
TicTacToe	9! = 362.880 (ca. 210.000 spielbar)	ja	ja
Mühle	ca 10^{10} relevante Stellungen	ja	ja
Vier gewinnt!	$\sim 10^{21}$ (ca. 10^{14} spielbar)	ja	-
Dame	$\sim 5,0 \cdot 10^{20}$	ja	ja
Awari	$\sim 10^{32}$, davon ca 10^{12} spielbar	ja	ja
Schach	$\sim 10^{120}$	nein	nein
Go (19×19 Feld)	$361! =\sim 1,7 \cdot 10^{727}$	nein	nein

Tabelle 3.1: Oberschranken der möglichen Spielstellungen für bekannte Spiele (entnommen aus [Zipproth 2003, Heule 2007, Wang/Y 2007, Allis 1988, Romein 2002, Gasser 1996]).

Suchbaum B_0^∞ zu kennen und abspeichern zu können. Offensichtlich überfordert dieser Ansatz die Möglichkeiten heutiger Rechenleistung für komplexere Spiele und dürfte bei Spielen mit der Komplexität von *Schach* und *Go* auch in absehbarer Zeit nicht gelingen. Die Tabelle 3.1 zeigt die Komplexität bekannter Spiele anhand der möglichen Spielstellungen.

Eine weitere Aussage über die Komplexität eines Spiels betrifft die Konvergenz von Spielen. Konvergierende Spiele besitzten die Eigenschaft, dass die Problemgröße im Laufe einer Partie, etwa durch eine immer geringere Anzahl möglicher Züge je Stellung, stetig abnimmt. Zur Lösung konvergierender Spiele wie etwa *Mühle* und *Awari* bieten sich Endspieldatenbanken an, die rückschreitend von terminalen Stellungen aus die Spielausgänge für mögliche vorhergende Stellungen speichern. Während einer Zugsuche wird dann vorwiegend nach einer in der Datenbank erfassten Stellung gesucht, ab welcher dann die ideale Bewertungsfunktion für die restlichen Folgestellungen vorliegt [Herik 2002, Allis 1994].

3.3.3 Gewichtete Stellungskriterien

Eine Bewertungsfunktion $Eval_s(S_t, P_i)$ untersucht für Spieler P_i einen Zustand nach dem Vorandensein von Stellungskriterien $k_1, k_2, \ldots, k_j$. Dafür steht der Bewertungsfunktion nur die Information über den Zustand S_t des Spielers P_s zur Verfügung. Jedem Kriterium k_i ist dabei ein Gewicht $w^i \in \overrightarrow{w}$ zugeordnet, das angibt, wie stark sich das Vorhandensein dieses Kriteriums auf die Gesamtaussage e_{p_i} auswirkt.

$$Eval_s(S_t, P) = \sum_{i=1}^{j} w^i \cdot K_i(S_t, P) \tag{3.2}$$

Die Kriteriumsfunktion $K_j(Q_t, P_i)$ liefert die Werte 0 oder 1 zurück, je nach-

dem ob das Kriterium k_i für Spieler P_i im Zustand S_t aus der Sicht des suchenden Spielers P_s vorhanden ist oder nicht. Als Untersuchungskriterien kommen häufig Siegpunkte und Spielentscheidende Muster in Frage. Je vollständiger die Kriterien alle für einen Sieg relevanten Bedingungen abdecken, desto präziser kann die Spielsituation eingeschätzt werden.

Eine ideale Bewertungsfunktion entspricht dabei der kompletten Erfassung aller entscheidenden Stellungskriterien mit dem richtig gewählten Gewichtsvektor.

3.3.4 Verfeinerung durch Stellungsklassen

Ein umfangreicher Satz an untersuchten Stellungskriterien führt zu einer kostenintensiveren Bewertungsfunktion. Dabei ist es gut möglich, dass einzelne Kriterien nicht bei jeder Spielstellung berücksichtigt werden müssen, da sie eventuell gar nicht mehr vorkommen können oder aus anderen Gründen belanglos für die genaue Einschätzung der Stellung sind.

Ein "Triggern" des Einsatzes der Kriterien kann über die Gewichte durchgeführt werden. Ein Gewicht $w^i = 0$ entspricht dabei der Deaktivierung des Kriteriums k_i. Dies würde bedeuten, den Gewichtsvektor von der untersuchten Stellung abhängig zu machen.

Da die Gewichtsvektoren für das Anpassen beim maschinellen Lernen abgespeichert werden müssen, folgt daraus, dass für jeden möglichen Zustand eines Spiels ein Speichereintrag vorliegt. Um diesen Schritt zu vermeiden, reduziert man den Raum S auf eine überschaubare Anzahl von n Stellungsklassen mithilfe eines dem Spieler eigenen Stellungsklassifizierer $C : S \rightarrow [1 \ldots n]$ [Block 2004].

Die Bewertunsgfunktion muss dafür um stellungsklassenabhängige Gewichtsvektoren erweitert werden.

$$Eval_s(S_t, P) = \sum_{i=1}^{j} w^i_{C(S_t)} \star K_i(S_t, P) \tag{3.3}$$

Dabei bezeichnet $w^i_{C(S_t)}$ die i.te Koordinate des Gewichtsvektors der Stellungsklasse des Zustandes S_t.

3.3.5 Normierung der Gewichtsvektoren

Da die Stellungskriterien mit den Werten 0 und 1 in die Bewertung eingehen, kann nur über eine Anpassung der Gewichte eine Normierung der Bewertungs-

vektoren durchgeführt werden. Diese sollte vorgenommen werden, damit $Eval$ nahtlos in v übergeht und somit die Lücke für nicht-terminale Zustände füllt. Also muss $Eval$ so normiert sein, dass es nur Ergebnisse aus dem Intervall $[-eval_{max}, eval_{max}]$ zurückliefert.

Die Normierungsbedingung für einen Gewichtsvektor $\overrightarrow{w_c}$

$$\sum_{i}^{\overrightarrow{w_c}} \left|w_c^i\right| = eval_{max} \tag{3.4}$$

erfüllt diese Bedingung des nahtlosen Übergangs. Wenn

$$Eval_S(S_t, P_a) > Eval_S(S_t, P_b)$$

wahr ist, so steht Spieler P_a aus Sicht des Spielers P_S, der die $Eval_S$-Funktion verwendet, einem möglichen Sieg näher als Spieler P_b.

3.3.6 Auswahl- und Erwartungsfunktion

Wurde ein Suchbaum B_t^d vollständig aufgespannt und wurden alle Blattknoten bewertet, so kann man daraus eine Erwartungsfunktion J_S für einen suchenden Spieler P_S ableiten, die für jeden Spieler P_i und jeden Knoten des Suchbaumes S_{t+j} angibt, welche Bewertung durch rationales Handeln aller nachfolgenden Spieler von diesem Knoten aus für P_i zu erwarten ist. Das rationale Vorgehen der Spieler ist dabei in einer Auswahlfunktion arg_f definiert, welche den Index des Zuges ermittelt, den der ziehende Spieler auswählt. Die Funktion J_S

$$J(S_{t+d}, P_i) = Eval_S(S_{t+d}, P_i) \tag{3.5}$$

in einem Suchbaum B_t^d entspricht an allen Blattknoten genau der Bewertungsfunktion $Eval_S$. Für innere Knoten S_{t+j} mit $0 \leq j < d$ von B_t^d bietet sich eine rekursive Definition

$$J(S_{t+j}, P_i) = J(S_{t+j} \circ a^{best}, P_i) \tag{3.6}$$

an. Der wahrscheinlich ausgewählte Zug a^{best} wird dabei durch die Auswahlfunktion

$$a^{best} = arg_f(J(S_{t+j} \circ a_i, a_{i \to P}))$$

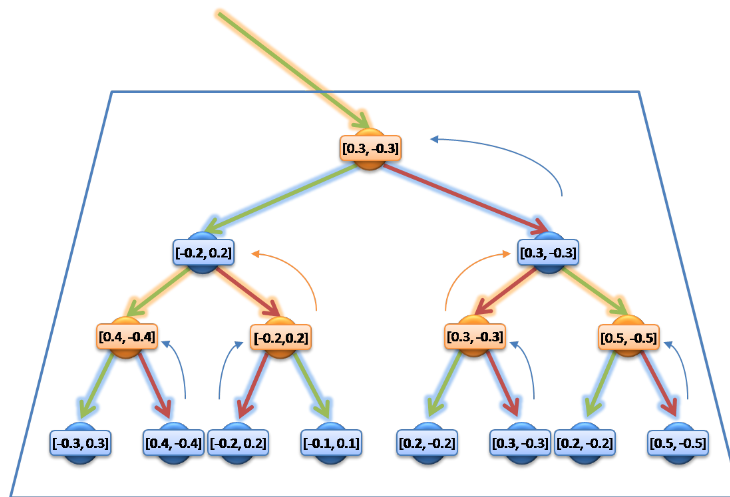


Abbildung 3.10: Die Erwartungswerte für alle Knoten im Suchbaum. Die Blattknoten werden mit der Bewertunsgfunktion des blauen Spielers gefüllt und dann anhand der Auswahlfunktion, hier arg_{max}, rekursiv über die rot gekennzeichneten besten Züge a^{max}nach oben weitergereicht.

ermittelt.

Eine der gängisten Auswahlfunktion ist die Maximierungsfunktion arg_{max}. Spieler versuchen dabei, wenn sie an der Reihe sind, denjenigen Zug auszuwählen, der den größten Erwartungswert verspricht. Dieser Ansatz erscheint für viele Spiele plausibel, führt jedoch in MS-Spielen zu einem Entscheidungsproblem (siehe Abschnitt 5.1). Zur Lösung des Problems wurden Algorithmen vorgeschlagen, die nur Änderungen an der Entscheidungsfunktion durchführen.

Neben der Bewertungsfunktion ist die Auswahlfunktion maßgeblich für das Spielverhalten verantwortlich. Spieler können Stellungen auf die gleiche Art bewerten und dennoch unterschiedliche optimale Züge finden, wenn sie unterschiedliche Auswahlfunktionen verwenden.

Die Angabe der Auswahlfunktion gehört zur eindeutigen Beschreibung der Spielweise eines Spielers. Ein aggressiver Spieler könnte zum Beispiel versuchen, immer den Zug auszuwählen, der ungeachtet des eigenen Erwartungswertes die Werte der Gegner minimiert. Je nach Spielart und gewünschtem Verhalten des Spielers bieten sich unterschiedliche Auswahlfunktionen an.

An probabilistischen Knoten kann ein Spieler nicht aktiv in das Spielgeschehen eingreifen und eine Wahl des Zufallereignisses treffen. Für einen solchen Knoten liefert J_S das statistisches Mittel

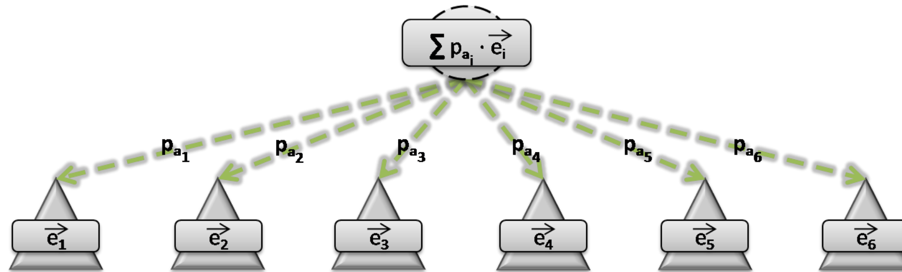


Abbildung 3.11: An Zufallsknoten wird dür den Erwartungswert das statistische Mittel der Erwartungen aller Zufallsereignisse verwendet.

$$J(S_{t+j}, P_i) = \sum_{i=1}^{b(t+i)} p_{a_i^*} \star J(S_{t+j} \circ a_i^*, P_i) \tag{3.7}$$

aller zu erwartenden Werte der Kinderknoten.

Auf Grund der Gleichung 3.1 ist für das statistische Mittel die Bedingung der Nullsummen-Erwartung erfüllt. In Abschnitt 7.2.4 wird besprochen, inwiefern die gewichtete Summe aller Erwartungswerte der geeignete Weg ist, Zufallsknoten zu handhaben.

Kapitel 4

Ein allgemeiner Suchalgorithmus

Ist die Erwartungswertfunktion J_S für jeden Knoten, jede Tiefe und jeden Spieler in einem Suchbaum B_t^d berechnet und zum Beispiel mit Hilfe dynamischer Programmierung gespeichert worden, so lässt sich eine allgemeine Suche wie folgt definieren:

```
// pre: Expectation[maxDepth][numPlayers][movesAtDepth]
// for current SearchTree B_{current}^{maxDepth}
findStrategy(state current, strategy sofar, numeric depth, player searchingPlayer)

 // die Strategie verläuft bis zu einem Blattknoten des Suchbaumes
 if depth > 0

  // alle von diesem Knoten ausgehenden Spielzüge oder
  // Zufallsereignisse aus der Sicht des Suchenden generieren
  actions = searchingPlayer->generateAllActions(current)

  // der Index des als nächsten ziehenden Spielers
  cp = current->currentPlayer()

  // wenn es sich um ein Zufallsknoten handelt
  if current is RandomState

   // werden für alle Zufallsereignisse
   for i : 1 to actions->size

    // neue Teilstrategie angelegt
    sofar->insertNode (actions[i])

    // der rekursive Aufruf für die Folgeknoten nach diesem Zufallsereignis
    findStrategy(current->newstate(actions[i]), sofar, depth - 1, viewpoint)
```

```
    // ansonsten handelt es sich um einen deterministischen Zufallsknoten
    else

     // und der beste Zug wird anhand der Auswahlfunktion und
     // den Erwartungswerten der Kindknoten ermittelt
     best = argf (Expectation, depth, CurrentPlayer])

     // dieser Zug wird der Strategie hinzugefügt
     sofar->insertChild(actions[best])

     // der rekursive Aufruf für die Folgeknoten nach diesem besten Zug
     findStrategy(current->newstate(actions[best]), sofar, depth-1, viewpoint)
Initialaufruf für Suche im Suchbaum B_t^d durch P_s:
Π = new strategy()
findStrategy(S_t, Π, d, P_s)
```

Eine Suche, die sich mit dem hier vorgestellten Schema formulieren lässt, nennt sich rational im Sinne der gewählten Auswahlfunktion. Die resultierende Strategie führt über die mit der Auswahlfunktion gefundenen besten Zügen zu den Blattknoten der Strategie und wird optimale Strategie aus Sicht des suchenden Spielers genannt.

Ein optimaler Spieler wählt zu jedem Zeitpunkt des Spiels denjenigen Zug, der die optimale Strategie aus seiner Sicht realisiert. An einem Zufallsknoten findet für jedes Zufallsereignis eine Zerlegung in Teilstrategien statt, die aus den jeweils besten Antwortzügen auf das Zufallsereignis bestehen.

4.1 Das strategische Nash-Gleichgewicht

Eine Strategie heißt Nash-Gleichgewicht, auch strategisches Gleichgewicht oder Nash-Cournot-Gleichgewicht, wenn alle Spielerzüge der Strategie die jeweils beste Antwort auf den vorgehenden Spielerzug oder das Zufallsereignis darstellen [Amann 1999]. Kein Spieler hat dabei einen Anreiz, einseitig von der optimalen Strategie abzuweichen, da er sich bei weiterhin optimalem Spielverhalten der Gegner dadurch verschlechtern würde. Die formale Definition und der Existenzbeweis wurden vom Namensgeber John Nash in seiner für die Spieltheorie zentralen Doktorarbeit 1950 eingeführt [Nash 1950].

Die hier vorgestellte optimale Suche in deterministischen 2S-Spielen findet die Strategie des strategischen Gleichgewichts des Suchbaumes. Dabei handelt es sich nicht zwangsläufig um das Nash-Gleichgewicht des gesamten Suchbaumes B_0^∞ oder gar des Spielbaumes G, da die Suche häufig nur einen durch die Suchtiefe d beschränkten Suchbaum behandeln kann.

Der Wert des Gleichgewichts entspricht des während der Suche nach der optimalen Strategie Π_t berechneten Erwartungswerts $\overrightarrow{e_{\Pi_t}}$ für den Ausgangszustandsknoten S_t.

Nash-Gleichgewichte lassen sich auf für komplexere Spiele mit Zufallselementen und unvollständiger Information definieren. Sie verlieren dabei jedoch an Aussagekraft über den erreichbaren Spielstand, da sie teilweise nicht spielbar oder auf Vermutungen und statistische Mittel zurückzuführen sind. Dennoch bieten strategische Gleichgewichte innerhalb des Spielerswissens die jeweils optimale Reaktion auf vorhergehende Spielerzüge und die aktuelle Spielsituation (vgl. [Amann 1999, Russel/Norvig 2004]).

4.2 Suchalgorithmen für 2S-Nullsummenspiele

4.2.1 MiniMax-Algorithmus

Der Minimax-Algorithmus realisiert eine rationale Suche für deterministische 2S-Nullsummen-Spiele mit Hilfe der $arg_{min/max}$ Auswahlfunktion und geht auf das für die Spieltheorie zentrale Minimax-Theorem zurück. Dieser durch von Neumann eingeführte Satz sagt die Existenz einer optimalen Strategie in allen 2S-Nullsummen-Spielen voraus und bildet somit die Grundlage für alle späteren Erweiterungen zu komplexeren Spielen [Neumann 1928].

Eine optimale Strategie wird gefunden, indem der jeweils ziehende Spieler immer mit dem stärksten Zug auf die aktuelle Spielsituation reagiert. Dies kommt einer alternierenden Abwechslung der Bewertungsmaximierung und Bewertungsminimierung durch die Auswahlfunktion $arg_{min/max}$ gleich, wovon sich der Name Minimax-Algorithmus ableiten lässt.

Mittels einer rekursiv definierten Tiefensuche können die Erwartungswerte innerhalb der Auswahlfunktion ermittelt werden. Dabei werden für alle Kinderknoten die Erwartungswerte im rekursiven Schritt ermittelt und miteinander verglichen. Je nach Auswahlschritt wird dann der größte oder kleinste Wert als Funktionswert zurückgeliefert. An terminalen Stellungen wird der reale Spielausgang zurück gegeben. An Blattknoten die Bewertung der erreichten Stellung.

Die Laufzeit des $MiniMax$-Algorithmus hängt direkt von der Anzahl der besuchten Knoten ab. Wobei für die durschnittlich b^d Blattknoten ein eventuell ungleich größerer Berechnungsschritt erfolgt, da hier die Bewertungsfunktion aufgerufen wird (Annahme: $T(Eval) \gg T(arg_f)$). In einem vollständigen Suchbaum B_t^d mit einem konstantem Verzweigungsfaktor b beträgt die genaue

Laufzeit daher:

$$\Theta(MinMax) = \Theta(b^d * T(Eval) + \sum_{i}^{d-1}(b^i * T(arg_f)) \tag{4.1}$$

Allgemein werden die Laufzeiten der rationalen Suchalgorithmen nur noch in Abhängigkeit der zu bewertenden Blattknoten angegeben, da hierfür die meiste Rechenzeit benötigt wird (Annahme: $T(Eval) \gg 1$). Diese Zahl wird häufig mit n angeführt. In Falle des MiniMax-Algorithmus gilt dann: $n = b^d$ und somit $\Theta(MiniMax) = \Theta(n)$ [Ertel 2008].

4.2.2 Pruning-Methoden

Ärgerlich an der $MiniMax$-Strategie ist die Tatsache, dass für die Berechnung des strategischen Gleichgewichts auch diejenigen Knoten berücksichtigt werden, die letztendlich nicht in der ermittelten Strategie enthalten sind. Abschneide-Methoden (*pruning methods*) versuchen, möglichst viele dieser irrelevanten Knoten und daran anknüpfende Teilbäume abzuschneiden und somit den Suchbaum zu verkleinern und die Anzahl der zu bewertenden Knoten zu minimieren.

Die zwei wichtigsten allgemeinen Pruning-Methoden sind [Sturtevant 2000]:

- sofortiges Abschneiden (*immediate pruning*): Führt eine vom Zustand S_t ausgehende Aktion a_i zu einem Knoten mit der maximalen Bewertung für den ziehenden Spieler, so müssen alle weiteren Züge a_j mit $j > i$ nicht mehr berechnet werden, da sich der ziehende Spieler hier nicht mehr verbessern werden kann. Dies ist dann der Fall, wenn die optimale Strategie am Zustand $S_t \circ a_i$ zu einem Sieg für den ziehenden Spieler führt (siehe Abbildung). Da die Häufigkeit siegreicher Strategien in Suchbäumen von der Suchtiefe um das Spiel abhängt, ist es nicht möglich, eine allgemeine Aussage zu treffen, wie stark diese Methode die Suche beschleunigt.

- verzweigtes und beschränktes Abschneiden (*branch and bound pruning*; in MS-Spielen: *shallow pruning*): Diese Methode erweitert das sofortige Abschneiden auch für nicht terminale Stellungen innerhalb des Suchbaumes. Besteht die Möglichkeit, dass durch einen Zug a_j mit $j > 1$, eine Verschlechterung für den ziehenden Spieler gegenüber dem bisherigen besten Erwartungswert $e_{tmp} = J_S(S_{t+j} \circ a_i, a_{i \to P})$ mit $a_i < j$ durch die Gegner mit optimalen Spiel erzwungen werden kann, so können alle noch nicht expandierten Teilbäume des Zustands S_{t+1} abgeschnitten werden. Es gibt

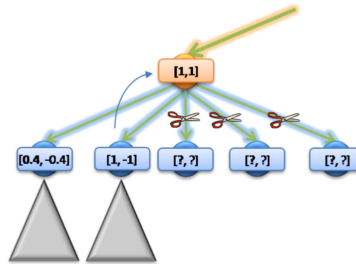


Abbildung 4.1: Ein Beispiel des sofortigen Abschneidens nach Vorfinden einer siegreichen Strategie.

keinen Grund für den ziehenden Spieler, Zug a_j zu spielen, da er bei optimalem Spiel der Gegner sich verschlechtern wird. Der Zug a_j muss daher nicht weiter in Betracht gezogen werden. Man spricht dabei, von einer durch e_{tmp} beschränkten Suche.

Weitere Pruning Methoden bieten sich für spezielle Problemklassen an und sind daher oft schwer zu verallgemeinern. So können neben den oben beschriebenen Schranken noch weitere gefunden werden, anhand derer sich Abschneidungen ableiten lassen.

Bei MS-Spielen mit einer beschränkten Maximalpunktezahl, wie etwa das Spiel *Hearts* ($sum_{max} = 26$) bietet sich beispielsweise das *deeppruning* an. Diese Methode nutzt die Tatsache, dass nach bereits verteilten Punkten sich neue Schranken im Suchbaum finden lassen, mit denen sich Abschneidungen vornehmen lassen [Sturtevant 2000].

Welche Aussage lässt sich über die Beschleunigung des *branch and bound pruning* treffen? Als Beispiel betrachten wir hier den Einsatz in einem 2S-Spiel mit einer Nullsummenbewertungsfunktion wie in der Abbildung 4.2 dargestellt. Wenn Spieler Blau den Zug a_3 als ersten betrachtet, ermittelt er eine eigene Bewertung von 0.4. Betrachtet er nun die weiteren Spielzüge a_2 und a_3, so stellt er fest, dass er bereits mit dem jeweils ersten Antwortzug auf diese Aktionen vom gelben Spieler in eine schlechte Position gezwungen werden kann. Daher kann er die Suche an dieser Stelle abschneiden und braucht keine weiteren Antwortzüge mehr zu berücksichtigen.

Die Sortierung der generierten Züge ist somit relevant für den besten Laufzeitfall eines *branch and bound pruning* Verfahrens. Dieser stellt sich genau dann ein, wenn der jeweils erste generierte Zug eines Knotens gleich auch der optimale ist. Für ein 2S-Suchalgorithmus, der diese Methode anwendet (auch

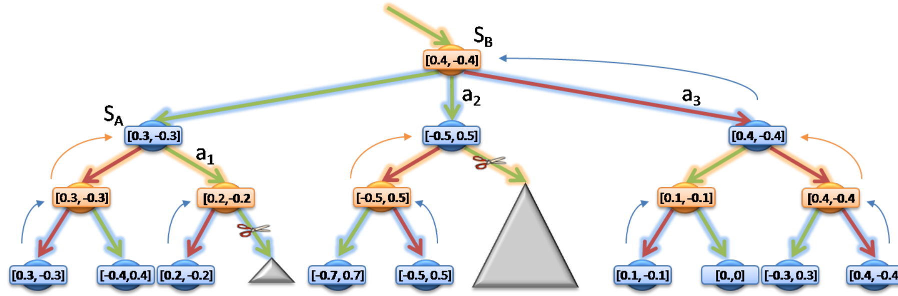


Abbildung 4.2: Beispiel für das Schranken abhängige Abschneiden. Bereits nach Betrachten des ersten Antwortzuges auf den Spielzug $a_{1.1}$ erkennt der gelbe Spieler, dass sein Gegner ihn bereits in eine schlechtere Position zwingen kann und braucht daher keine weiteren Antwortzüge zu berücksichtigen. Gleiches gilt für den blauen Spieler nach Betrachten des ersten Antwortzuges auf a_2. Eine günstige Sortierung (etwa a_3 an erster Stelle) hätte Spieler Blau weitere Abschneidungen und somit eine schnellere Suche ermöglicht.

alpha-beta-pruning genannt), wird der Verzweigungsfaktor b durschnittlich auf $\sqrt{b}$ reduziert. Somit wird die Laufzeit des MiniMax-Algorithmus auf

$$\Theta(AlphaBeta) = \Theta(\sqrt{b}^d) = \Theta(\sqrt{n})$$

reduziert [Ertel 2008].

Formal bedeutet dies, dass die zum Nash-Gleichgewicht führende Strategie bereits im ersten von der Suche bewerteten Blattknoten erreicht sein muss. Wenn dies nicht der Fall wäre, würde dies bedeuten, dass an einer Stellung nicht der erste Zug zum Nash-Gleichgewicht geführt hat. Dies steht im Wiederspruch zur Annahme, dass der erste Zug bei perfekter Zugsortierung der optimale sein muss.

4.2.3 Zugsortierungsheuristiken

Eine Reduktion der Laufzeit bedeutet bei fester Suchzeit immer auch tiefere Suche und somit eine wohl stärkere Antwort auf die aktuelle Spielstellung. Probleme, die durch die Reduktion des Spielbaums auf einen endlichen Suchbaum entstehen können, werden dadurch minimiert. Da eine perfekte Zugsortierung ähnlich wie $Eval^*$ schwer zu finden sein dürfte, muss man auch hier eine Heuristik $score_f \ : \ A_t \rightarrow [1 \ldots |A_t|] \times \mathbb{R}$ anwenden.

Diese versieht alle möglichen ausgehenden Züge eines Knotens mit einem *score* $(a_{i \rightarrow score})$, anhand dessen die Sortierung absteigend vorgenommen wird.

Der *score* wird dabei aus der Sicht des suchenden Spielers für den ziehenden Spieler ermittelt.

Idealerweise versucht eine gute Sortierungsheuristik die *score*-Funktion den Werten von $Eval^*$ anzunähern. Diese Funktion würde immer den stärksten Zug an erster Stelle positionieren. Das Nash-Gleichgewicht wäre dann beim ersten Blattknoten gefunden und eine maximale Anzahl an Abschneidungen könnte durchgeführt werden.

Offensichtlich ist dieser Ansatz nur bedingt möglich und heuristische Konzepte müssen gefunden werden:

- $score_{Eval_S}$: Für die ausgehenden Züge a_i eines Knotens S_t gilt: $a_{i \to score} = Eval_s(S_t \circ a_i, a_{i \to P})$. Dieser Ansatz entspricht am ehesten dem menschlichen Verhalten. Es handelt sich um eine zusätzliche Suche mit Suchtiefe 1 an jedem Knoten. Die besten Folgestellungen werden an den Anfang der Zugliste gestellt.

- $score_{J_S}$: Für die ausgehenden Züge a_i eines Knotens S_t gilt: $a_{i \to score} = J_S(S_t \circ a_i, a_{i \to P})$. Vorteil ist hierbei, dass ein wesentlich tieferer Blick in die Zukunft die Sortierung der Züge bestimmt und somit mögliche Horizonteffekte (vgl. Abschnitt 7.2.1) vermieden werden können. Nachteil ist, dass die Erwartungswerte für alle Knoten des Suchbaumes ja eigentlich erst während der Suche berechnet werden und diese Werte im bisherigen Suchalgorithmus nicht vorliegen. Dies kann man mit Tiefeniteration umgehen, indem mit dynamischer Programmierung die Erwartungswerte der Knoten aus vorhergehender Suchen als *score* für die neue, tiefere Suche verwendet werden (siehe Abschnitt 4.4.2).

- Den *score* von Zügen aus einer Tabelle entnehmen: dieser Ansatz eignet sich gut für frühe Spielphasen, da hier noch ein überschaubarer Raum von möglichen Zuständen vorliegt. Sogenannte Eröffnungsbücher lassen sich durch Analyse vieler bekannter Partien erstellen [Steinwender 1995]. Zu jedem im Eröffnungsbuch erfassten Zustand berechnet man für die möglichen ausgehenden Züge eine Wahrscheinlichkeit, mit der sie real gespielt wurden. Handelt es sich bei den verwerteten Partien um Partien starker Spieler, so lassen sich mit diesem Ansatz gute Vorhersagen über das Spielverhalten der Gegner erstellen. Sobald selbst der wahrscheinlichste Zug einer Stellung einen festgelegten Schwellwert unterschreitet (z.B $a_{i \to score} < \frac{5}{\tau}$, bei $\tau > 5$), verlässt man das Eröffnungsbuch und wendet

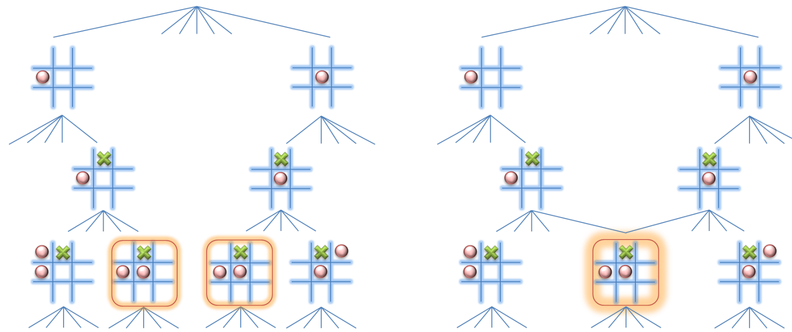


Abbildung 4.3: Sich wiederholende Knoten verschmelzen zu einem. Aus dem Suchbaum wird ein Suchgraph.

andere Sortierungsheuristiken an.

- $score_f$-Funktion maschinell lernen: Dafür werden Züge klassifiziert und ihrem Zugtyp nach mit unterschiedlichen *score*-Werten versehen. So könnten sich im Schach die Schlag- und Schachzüge häufiger am Anfang der Zugliste befinden, da sie womöglich eher zum strategischen Gleichgewicht führen als ein harmloser Bauernzug. Neben der Zugklassifizierung kann auch eine Gewichtung der Zugtypen erlernt werden, idealerweise noch in Abhängigkeit der aktuellen Stellungsklasse.

Eine Zugsortierung wird vorerst nur an deterministischen Knoten vorgenommen, da nur hier Abschneidungen durchgeführt werden können. In Abschnitt 5.4.2 werden Ansatzideen besprochen, mit denen man auch an Zufallsknoten die Suche mittels einer geeigneten Sortierung der Zufallsereignisse beschleunigen kann.

Die verwendete Zugsortierung ist spielerabhängig und wird somit nicht im allgemeinen Zuggenerator realisiert. Der Suchalgorithmus greift für das Aufspannen des Suchbaumes auf die vom suchenden Spieler sortierte Zugliste zurück.

4.3 Transpositionstabellen

In den meisten Spielen können sich Zustände im Verlauf einer Partie wiederholen. So kann es etwa beim Schach zu Brettstellungen kommen, die bereits zu einem früheren Zeitpunkt eingetreten sind. Solche Wiederholungen müssen teilweise in manchen Spielen allein schon aufgrund der Spielregeln erkannt werden.

Die redundante und unnötige mehrfache Berechnung von Knoten kann mit Hilfe von sogenannten Transpositionstabellen vermieden werden. Eine Transpositionstabelle ist eine Hashtabelle, die für einen eindeutigen Schlüssel einer Stellung weitere Informationen abspeichert, auf die man bei erneutem Vorfinden der Stellung zurückgreifen kann (u.a. in [Marsland 1986]).

4.3.1 Zobrist-Schlüssel

Hash-Tabellen mit einzigartigem Schlüssel für jede unterscheidbare Eingabe verhindern Schlüssel-Kollisionen und erreichen die optimale Laufzeit aller Operationen von $O(1)$ [Goodrich 2002].

Eine geeignete Schlüsselerzeugung, die für beliebige Zustandsräume fast eindeutige Schlüssel generiert, wird in [Zobrist 1970] vorgestellt. Darin wird gezeigt, wie sich die Wahl einer Schlüssellänge auf Hash-Kollisionen zweier unterschiedlicher Stellungen auswirkt und wie diese minimiert werden können.

Der Einsatz von Zobrist-Schlüsseln wurde unter anderem erfolgreich in der Schachprogrammierung getestet (z.B. in [Block 2004, Warnock 1988]) und lässt sich einfach für weitere Spiele erweitern. Es ist die Aufgabe der Entwickler von Spielen, das Konzept der Zobrist-Schlüssel für das neu eingebundene Spiel zu realisieren.

Für jede neu in die Transpositionstabelle aufgenommene Stellung S_t werden unter dem Schlüssel $Zobrist(S_t)$ mehrere Werte gespeichert:

- Der Erwartungswertvektor $\overrightarrow{e}$ für S_t
- Knotentyp: handelt es sich um exakte Erwartungswerte, eine untere oder obere Schranke?
- Die Suchtiefe d, an dem dieser Wert innerhalb des Suchbaumes ermittelt wurde
- Optional: der von S_t ausgehende optimale Folgezug a_{best}
- Optional: die Zugliste A_t

4.3.2 Varianten von Transpositionstabellen

An mehreren Stellen des Suchbaumes lassen sich Verbesserungen vornehmen [Schaeffer 1989]:

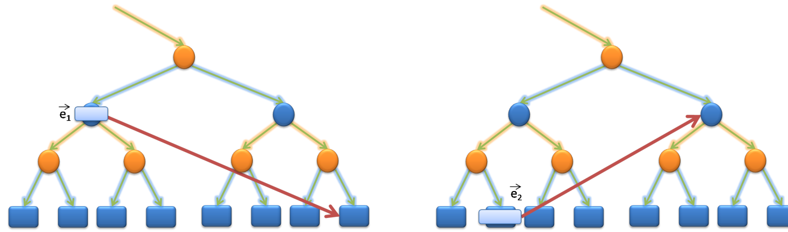


Abbildung 4.4: *Links*: Der Erwartungswertvektor $\vec{e_1}$ kann für die wiedergefundene Stellung verwendet werden, da er eine höhere Entfernung zu einem Blattknoten besitzt. *Rechts*: Hier kann der Vektor $\vec{e_2}$ nicht wieder verwendet werden, da $\vec{e_2}$ die Folgeknoten der wiedergefundenen Stellung nicht berücksichtigt. Eine Aktualisierung des in der Transpositionstabelle gespeicherten Wertes erfolgt nach vollständiger Berechnung der Folgestellungen.

- *Blattknoten:* Viele der erreichbaren Blattknoten entsprechen identischen Stellungen. Da die Bewertungsfunktion bei komplexen Spielern recht zeitaufwendig werden kann, lohnt es sich, mehrfache Funktionsaufrufe von *Eval* zu verhindern und statt dessen aus der Transpositionstabelle abzulesen.

- *Zuggenerierung*: Für jeden erreichten inneren Knoten wird die ausgehende Zugliste abgespeichert und bei erneutem Vorfinden des Knotens wieder verwendet. Diese Beschleunigung eignet sich vor allem für die iterative Tiefensuche, da hier alle Knoten bis zur Suchtiefe $d-1$ in der Folgesuche erneut erreicht werden.

- *Erwartungswerte*: Die Erwartungswerte innerer Knoten werden inklusive Knotentyp und Suchtiefe abgespeichert. Dabei dürfen nur Erwartungswerte von Knoten mit einer größeren oder gleichen Entfernung zur Wurzel übernommen werden, da sonst die Knoten nicht entsprechend ihrer Suchtiefe bewertet werden (siehe Abbildung 4.4). Bei der Wiederverwendung eines Erwartungswertes muss ebenfalls berücksichtigt werden, um welchen Knotentyp es sich handelt. Wurden Abschneidungen durchgeführt, kann es sein, dass der Erwartungswert nicht dem exakten berechneten Wert, sondern einer oberen oder unteren Schranke entspricht.

Anmerkung: In [Breuker 1995] werden unterschiedliche Schemata vorgestellt, wie man bereits vorhandene Einträge in den Transpositionstabellen mit neuen verknüpft. Der hier vorgestellte Ansatz entspricht dem *deep replacement* Schema, nach dem der Erwartungswert derjenigen Stellung, die am weitesten entfernt

von einem Blattknoten berechnet wurde, bevorzugt in der Tabelle abgespeichert wird.

Andere Schemata sind allerdings auch denkbar, so zum Beispiel das *big replacement* Verfahren, welches Erwartungswerte von Teilbäumen bevorzugt in der Tabelle abspeichert, die eine größeren Menge an Knoten berücksichtigt haben.

4.4 Erweiterungen zu allgemeinen Spielklassen

Mit jeder hier vorgestellten Erweiterung wird eine größere Klasse von Spielen umfasst, für die der Suchalgorithmus eine Lösung findet, ohne die vorhergehende Klasse zu verlieren. Es wird also ein einziger Algorithmus entwickelt werden, mit dem alle Problemstellungen gelöst werden können. Beginnend bei Nullsummenspielen für zwei Spieler mit vollständiger Information ohne Zeitvorgabe, wird der Algorithmus nach und nach für folgende Spielarten erweitert:

- Nicht-Nullsummenspiele
- Suchzeitbegrenzungen
- Spiele mit Zufallsereignissen
- Spiele mit unvollständiger Information und verdeckten Zügen
- Nicht rundenbasierte Spiele

Die Menge aller möglichen Spielarten lässt sich nicht hierarchisch einteilen, da Spiele mit jeder möglichen Kombination dieser Erweiterungen denkbar sind.

4.4.1 Nicht-Nullsummenspiele

Der $MiniMax$-Algorithmus geht von der Annahme aus, dass der erste Spieler seinen Evaluationswert maximiert und der Gegenspieler seinen Wert minimiert. Diese Annahme ist richtig, wenn an die Bewertungsfunktion die Anforderung gestellt wurde, dass bei einem 2S-Nullsummen-Spiel Stellungskriterien zugunsten des ersten Spielers positiv, zugunsten des zweiten Spielers negativ in die Bewertung einfließen. Als Ergebnis wurde nur die Bewertung aus der Sicht eines Spielers berechnet und für den weiteren Spieler das negierte Ergebnis verwendet.

Beim Schach bedeutet dies zum Beispiel, nur die Bewertung für den weißen Spieler durchzuführen und für den schwarzen Spieler diesen Wert negiert zu verwenden. Hier tritt der Begriff des rationalen Handelns der Gegner auf:

Während der Partie nimmt man an, dass gegnerische Spieler Spielsituationen mit der gleichen eigenen Bewertungsfunktion analysieren und darauf aufbauend ihre Entscheidungen treffen. Rationales Handeln des Gegners heisst also, vorrübergehend die Seiten zu wechseln und das Spiel aus dieser Sicht zu lösen.

Es ist aber wichtig zu verstehen, dass Schach nicht zwangsläufig als Nullsummenspiel interpretiert werden muss. Zwar ist der Spielausgang eindeutig eine Nullsummensituation, dennoch ist in der Wahrnehmung der Spieler eine Partie zwischenzeitlich oft nicht mehr ein Nullsummenspiel. So können sich beide Spieler zum Beispiel gleichzeitig am Siegen sehen. Diese unterschiedliche Einschätzung ein und derselben Spielsituation rührt daher, dass Spieler Stellungen unterschiedlich bewerten und womöglich sogar unterschiedliche Auswahlfunktionen, also unterschiedliche Suchalgorithmen verwenden. In der Suche allerdings werden die Stellungen nur mit einer einzigen Bewertungsfunktion, nämlich der des Suchenden untersucht.

Angenommen ein Computerspieler weiss, dass sein Gegner bestimmte Stellungskriterien bevorzugt, die der Computerspieler selbst als nicht so vorteilhaft einstuft. Bewertet nun der Computerspieler die gegenerische Position mit der eigenen Bewertungsfunktion, so entsteht zwangsläufig ein Fehler, da hier das Wissen um die bevorzugten Stellungskriterien nicht einfließt. Dieser Fehler liegt in der Größenordnung der unterschiedlichen Bewertung des Stellungskriteriums, der Differenz zweier Parameter. Sobald Wissen über die Einstufung des Gegners für Stellungskriterien vorliegt, lohnt es sich, statt der eigenen einen an dieses Wissen angepassten Parametersatz für die Bewertungsfunktion zu verwenden.

Ein Gegnerprofil bestünde initial aus dem gleichen eigenen Parametersatz, welcher dann nach Trainingspartien nach und nach an das Verhalten des Gegners angepasst werden kann. Ein solches Vorgehen führt zu einer Verlangsamung der Suche, da nun statt wie bisher nur eine Bewertung einer Stellung für 2S-Spiele gleich zwei Durchgänge notwendig sind.

Als weitere Konsequenz folgt, dass die Summe beider Bewertungen nicht mehr zwangsläufig 0 ergibt. Dies behebt man, indem die Abstände beider Bewertungen zueinander als letztendliche Bewertung aus Sicht der jeweiligen Spieler verwendet werden:

$$[e_{p_1} \mid e_{p_2}] \Rightarrow [(e_{p_1} - e_{p_2}) \mid (e_{p_2} - e_{p_1})]$$

Mit der neuen Interpretation des Bewertungsvektors lassen sich auch alle Computerspieler einbinden, deren Bewertungsfunktion von vornerein keine

Nullsume erzeugte. Dies ist zum Beispiel bei einem Computerspieler für *Siedler von Catan* der Fall, der sich nur nach den erreichten Siegpunkten der Spieler orientiert und diese als Bewertung der Spielstellung zurückliefert.

Verzichtet man auf die triviale Gegnermodellierung, so behält die Bewertungsfunktion dennoch ihre Gültigkeit; die Werte verdoppeln sich lediglich:

$$[e_{p_1} \mid (-e_{p_1})] \Rightarrow [(e_{p_1} + e_{p_1}) \mid (-e_{p_1} - e_{p_1})]$$

Die Entscheidung bleibt demjenigen Spieler überlassen, der weiss, ob seine Bewertungsfunktion einen Nullsummencharakter besitzt oder nicht. Ein weiterer Grund, sich gegen die Erweiterung zu entscheiden, liegt in der nun zweifach ausgeführten Bewertungsfunktion, welche die Suche verlangsamt. Die Vorteile des Verfahrens zeigen sich aber deutlich bei Spielen mit unvollständiger Information, wie etwa *Poker* und *Siedler von Catan*. Bei MS-Spielen ist man sowieso gezwungen, die Bewertungswerte aller Spieler zu berechnen, so dass sich diese Erweiterung hier nicht mehr negativ auf die Suche auswirkt. (siehe Abschnitt 5).

4.4.2 Tiefeniteration und Zeitmanagement

In manchen Spielen existiert eine Vorgabe für die zur Verfügung stehende Suchzeit. Dabei kann jeder einzelnen Zugsuche eine zeitliche Grenze gesetzt sein, oder die Spielzeit für die komplette Partie ist begrenzt. Ein allgemein gehaltener Suchalgorithmus muss dies flexibel handhaben können. Dafür bietet sich das Verwenden einer iterativen Suche an, die nacheinander bis zu d Suchen mit den Suchtiefen $1, 2, ..., d$ durchführt. Nach jedem Durchgang wird überprüft, ob noch ausreichend Zeit für eine weitere Suche zur Verfügung steht. Ist dies nicht der Fall, liefert man den besten Zug der zuletzt durchgeführten Suche.

Der Ansatz bringt den Vorteil mit sich, dass ab einer Suchtiefe größer als 2 sich die Werte der $score_{J_S}$-Funktion ohne zusätzlichen Rechenaufwand für die inneren Knoten der nächsten Suchiteration ermitteln. Die Erwartungswerte werden in der Transpositionstabelle mitgeführt.

Der zusätzliche Rechenaufwand wird durch den effektiven Einsatz aller Transpositionstabellen stark minimiert. Um möglichst alle gewonnenen Erkenntnisse vorhergehender Suchen zu behalten, bietet es sich an, die Zustände als Knotenobjekte des Suchbaumes in den Transpositionstabellen abzuspeichern.

Es kann sogar gezeigt werden, dass der zusätzliche Rechenaufwand in einem

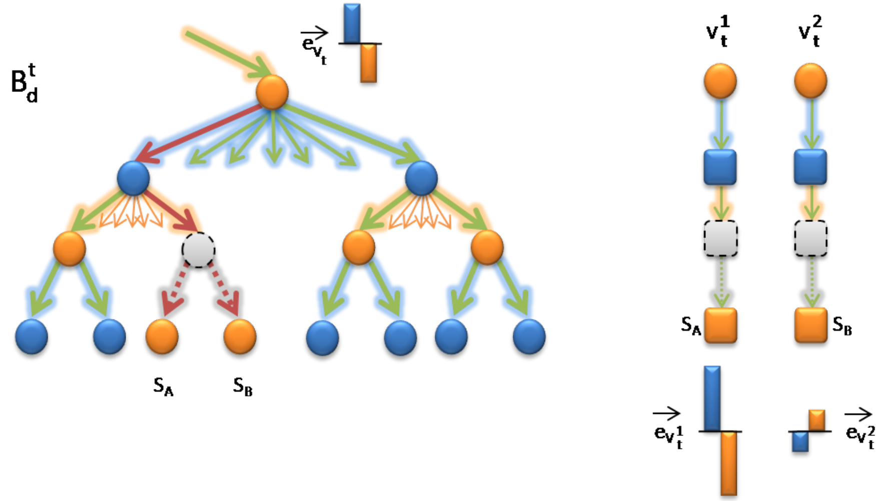


Abbildung 4.5: Die beiden Entscheidungsstrategien v_t^1 und v_t^1 der optimalen Strategie im Suchbaum B_d^t. Gemittelt über die Wahrscheinlichkeiten $p_{v_t^1}$ und $p_{v_t^2}$ ergibt sich aus den Erwartungswerten $\overrightarrow{e_{v_t^1}}$ und $\overrightarrow{e_{v_t^2}}$ der Gesamterwartungswert $\overrightarrow{e_{V_t}}$ für die optimale Strategie.

Algorithmus mit branch and bound pruning ausfällt, wenn dieser eine geeignete Zugsortierung realisiert [Ertel 2008]. Eine solche ist mit der $score_{J_S}$-Funktion gegeben.

4.4.3 Entscheidungsvarianten der optimalen Strategie

Als Entscheidungsvarianten eines Strategiebaumes bezeichnet man alle Pfade von der Wurzel zu den Blattknoten des Strategiebaumes. Dabei entsprechen die Blattknoten den Stellungen, die letztendlich für die Entscheidung während der Suche nach der optimalen Strategie im Mittel ausschlaggebend waren.

Eine Variante v_t der Länge j besteht aus einer Folge von Zügen und Zufallsereignissen $a_{1,}^v a_2^v \ldots, a_j^v$, der dadurch erreichten Stellung S_{t+j}, der Gesamtwahrscheinlichkeit $p_{v_t^j}$ und dem Bewertunsgvektor $\overrightarrow{e_{v_t^j}}$ der Stellung S_{t+j}. Strategien bestehen somit aus einer oder mehreren (n) Entscheidungsvarianten, welche zusammengefasst mit der Variablen $V_t = v_t^1, v_t^2, \ldots, v_t^n$ wiedergegeben werden.

Die Stellungsklasse k einer Entscheidungsvariante v_t entspricht der Stellungsklasse des erreichten Zustandes S_{t+j}. Dies wird durch die Funktion $C'(v_t) = k$ ausgedrückt.

Da eine Strategie für alle Zufallsereignisse eines Zufallsknotens eine Teilstrategie besitzt, ist die Summe der Wahrscheinlichkeiten aller Teilstrategien

immer gleich 1. Dies bedeutet, dass die Summe der Wahrscheinlichkeiten aller Entscheidungsvarianten ebenfalls 1 ergibt.

Entscheidungsvarianten sind ein wichtiges Werkzeug für die Anpassung der Gewichtskoeffizienten, wie sie in Kapitel 6 beschrieben werden. Hat sich eine Strategie im Nachhinein als doch ungünstig erwiesen, so kann man über die Entscheidungsvarianten dieser feststellen, welche Bewertungen korrigiert werden müssen, um eine bessere Einschätzung zu ermöglichen.

Zur besseren Unterscheidung werden die Zustände einer Variante rechteckig gezeichnet.

Kapitel 5

Die MS-Spielsuche $UPP - max^n$

Einige grundlegende Entscheidungen beim Suchverfahren müssen getroffen werden, wenn man dieses für MS-Spiele erweitern möchte. Die bisher verwendete Auswahlfunktion arg_{max} birgt einige Risiken, da sie nicht die Entwicklung der Gegenspieler und den Nutzen unterschiedliche Bewertungsvektoren für den suchenden Spieler berücksichtigt.

Die Problematik kann man deutlich am Beispiel der Abbildung 5.1 erkennen. Verwendet man die nicht normierten Bewertungsvektoren, so würde sich Spieler 1 bei einer reinen Maximierung der eigenen Position sich eher für einen Knoten mit der Bewertung $\overrightarrow{e_b}$ entscheiden, dabei aber nicht erkennen, dass er gegenüber einer Bewertung $\overrightarrow{e_a}$ relativ zu allen Gegnern verloren hat (von der dritten zur letzten Position).

5.1 Das Mehrspieler-Entscheidungsproblem MES

Auch eine für MS-Spiele erweiterte Normierung des Erwartungswertvektors löst diesen Konflikt nicht zwangsläufig. Aus Sicht des ersten Spielers gibt es auch dann keinen Unterschied zwischen $\overrightarrow{e_a}$ und $\overrightarrow{e_b}$, wenn nur der eigene Wert betrachtet wird. Gegen S_a spricht die oben genannte Verschlechterung gegenüber den Gegnern. Stellung S_b allerdings liegt einer maximalen Bewertungsposition scheinbar näher und könnte somit eventuell doch zu bevorzugen sein.

Der Verlust der eindeutigen Unterscheidbarkeit zweier Erwartungswertvek-

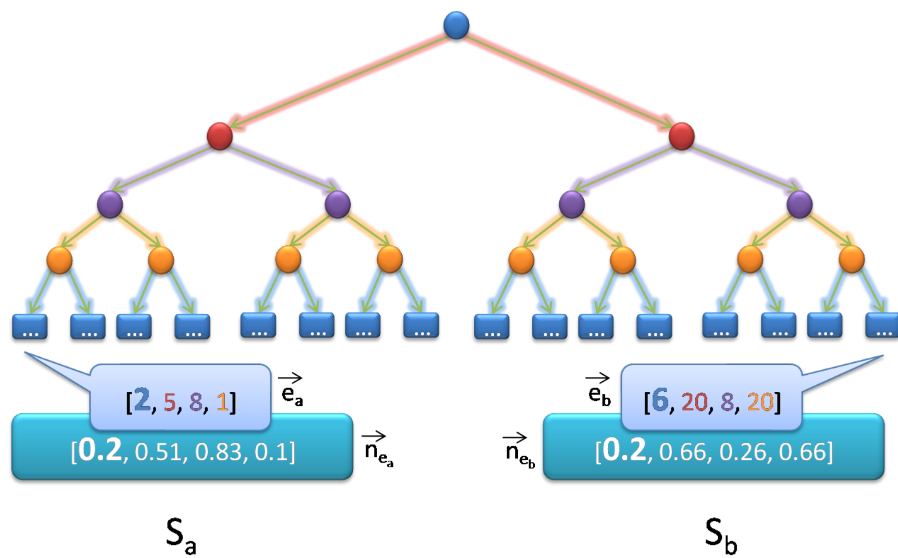


Abbildung 5.1: Beispiel für einen MS-Suchbaum: Aus der Sicht des ersten Spielers ist nicht mehr eindeutig, welcher der beiden Bewertungsvektoren zu bevorzugen ist, wenn die Entwicklung der Gegner und das Verhältnis der eigenen Position zu diesen nicht berücksichtigt werden. Eine reine Maximierung des Bewertungswertes würde dabei die scheinbar ungünstige Spielsituation S_b bevorzugen. Auch die Normierung des Vektors löst den Konflikt nicht.

toren ist das Mehrspieler-Entscheidungsproblem. Die Eindeutigkeit im verwendeten Suchalgorithmus ist nicht mehr gegeben und eine objektiv optimale Strategie von der Einstellung der Spieler zueinander abhängig. In realen Spielsituationen lösen Menschen diesen Konflikt, indem sie Kooperationen eingehen, anonym oder offen ausgesprochen.

Die Bevorzugung eines der beiden Vektoren $\vec{e_a}$ und $\vec{e_b}$ hängt dann davon ab, welche Spieler miteinander kooperieren. Liegt beispielsweise eine Allianz aller Spieler gegen Spieler violett vor, ist die Situation S_b auf einmal eindeutig die günstigere für Spieler blau, wenn er im Sinne der Allianz handelt.

Ist eine Allianz nicht in den Spielregeln wie etwa beim Kartenspiel Skat verankert, sondern nur durch mündliche Vereinbarungen enstanden, so ist das Eingehen einer solchen gefährlich. Zu groß ist das Risiko, dass die Allianz zum eigenem Vorteil von einem verbündeten Spieler gebrochen wird.

Besitzt beispielweise ein Spieler die Wahl zwischen zwei Zügen, wovon einer den eigenen Sieg und der andere dem eines Allianzpartners entspricht, so entscheidet er sich für den eigenen Sieg und übergeht die Allianz, wenn durch die Spielregeln nicht gesichert ist, dass er aus beiden Situationen den gleichen Vorteil bezieht. Es ist daher gefährlich, auf Allianzen zu vertrauen.

5.2 Mehrspieler-Suchalgorithmen

Eine alternative Art den Minimax-Algorithmus zu definieren ist, dass beide Spieler die Bewertung aus ihrer Sicht jeweils maximieren wollen. Dafür wird der Erwartungswert des zweiten Spielers einfach negiert und das Problem mit Hilfe einer reinen arg_{max}-Auswahlfunktion gelöst. Dies entspricht dem Konzept des Negamax-Algorithmus (vgl. Abschnitt 4.4.1).

5.2.1 Der max^n-Suchalgorithmus

Für MS-Spiele wurde die Idee des Negamax-Verfahrens auf n Spieler erweitert und der allgemeine max^n-Algorithmus entwickelt [Luckhardt 1986]. Dabei versuchen alle Spieler, die Bewertung aus ihrer Sicht innerhalb eines max^n-Suchbaumes zu maximieren.

Ein max^n-Suchbaum enthält an jedem Zustandsknoten Erwartungswertvektoren mit nur positiven Werten. Diese Werte entsprechen der Bewertung aus der Sicht des zu bewertenden Spielers. Zur besseren Darstellung der Spielsituation werden alle Werte abzüglich des Durchschnittswerts angezeigt. Dadurch

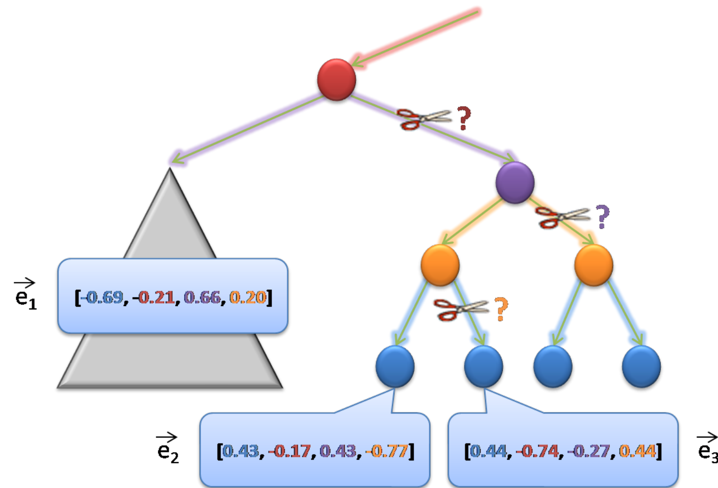


Abbildung 5.2: Obwohl sich die Spieler Rot, Violett und Orange gegenüber der Erwartung des linken Teilzugs $\vec{e_1}$ in $\vec{e_2}$ verschlechtern, ist ein Abschneiden solange nicht möglich, bis sie wissen, ob Spieler Blau sich doch nicht für die Bewertung $\vec{e_3}$ entscheidet. Spieler Orange hätte die letztendlich von Spieler 2 gewählte Bewertung $\vec{e_3}$ übersehen, die für beide eine Verbesserung gegenüber $\vec{e_1}$ darstellt.

erscheinen führende Spieler positiv, zurückliegende negativ. Demnach ist der Negamax-Algorithmus ein max^2-Algorithmus.

Leider lassen sich in max^n Suchbäumen mit $n > 2$ nicht mehr wie gewohnt *branch and bound pruning* Methoden anwenden. Dies wird in Abbildung 5.2 deutlich. Sobald die Bewertung $\vec{e_2}$ vorliegt, könnten alle 3 Gegenspieler von Spieler Blau gewillt sein, Abschneidungen durchzuführen, da sie erkennen, dass dieser sie in eine schlechtere Position gegenüber $\vec{e_1}$ führen wird. Sie besitzen allerdings keine Garantie, dass Spieler Blau diesen Zug auswählen wird, wenn sie nicht auch die Bewertung $\vec{e_3}$ ermitteln. Hier zeigt sich, dass Spieler Orange sich sogar noch verbessern kann, da Spieler Blau, seine Erwartung maximierend sich für den zweiten Zug entscheiden wird.

Ebenfalls können Spieler Rot und Violett noch keine Abschneidungen durchführen, obwohl alle bisher im rechten Teilbaum vorgefundenen Bewertungen eine Verschlechterung darstellen. In den beiden letzten noch ausstehenden Blattknoten könnte ja noch eine Verbesserung, die auch von Spieler Blau und Orange mitgetragen wird, vorhanden sein.

Der max^n-Algorithmus umgeht das MS-Entscheidungsproblem, indem er einfach die Entwicklung der Gegner ausser Acht lässt. Diese Annahme bietet

sich in vielen Spielen an, die auch für Mehrspieler einen Nullsummencharakter, etwa beschränkte Punkte oder Ressourcen, besitzen. Die prinzipielle Entscheidungsschwierigkeit ist dadurch aber nicht behoben und ist mit der Grund dafür, dass die bisherigen Abschneide-Methoden zur Beschleunigung der Suche scheitern.

Eine Art des Abschneidens wird dennoch ermöglicht, wenn eine maximale Bewertungssumme $max_{eval-sum}$ vorliegt. Bei n Spielern eines nicht-Nullsummenspiels wäre dies $n \star eval_{max}$. Konnte der bei S_t ziehende Spieler P_i sich im ersten Teilbaum von S_t bereits einen festen Wert e_i sichern, so besitzen die restlichen Spieler in den weiteren Teilbäumen nur noch maximal die Möglichkeit, den Wert $e_{maxSum \setminus P_i} = max_{eval-sum} - e_i$ zu erzielen. Dies rührt daher, dass ein höherer Wert als $e_{maxSum \setminus P_i}$ für P_i eine Verschlechterung bedeuten würde und er dies daher verhindert. Alle weiteren noch nicht expandierten Folgezüge können dann abgeschnitten werden.

Für max^n-Suchalgorithmen konnte bewiesen werden, dass sich im besten Fall des $maxSum$-prunings durch einen günstigen Spielverlauf und eine optimale Zugsortierung der durchschnittliche Verzweigungsfaktor b nur auf den Faktor $\frac{1+\sqrt{4b-3}}{2}$ verringern ließ. Schlimmer noch, der mittlere Fall ergab, dass asymptotisch der Faktor überhaupt nicht reduziert werden kann [Korf 1991].

5.2.2 Der *paranoid*-Algorithmus

Der in [Sturtevant 2000] vorgestellte *paranoid*-Suchalgorithmus bietet einen alternativen Lösungsweg an. Hier wird die "paranoide" Annahme getroffen, dass sich alle $(n-1)$ Gegenspieler gegen den suchenden Spieler verbündet haben und zu jedem Zeitpunkt versuchen, dessen Position zu minimieren, ungeachtet der eigenen Entwicklung. Das Spiel wird somit auf eine "Ich gegen alle"-Variante reduziert.

Die Suche entspricht dann wieder einer 2S-Situation, für welche die Optimierungen aus Kapitel 4 angewandt werden können. Der Erwartungswert des suchenden Spielers wird als Schranke für das *branch and bound pruning* angewendet. Das Verfahren ermöglicht eine drastische Reduktion der untersuchten Knoten in einem Suchbaum auf bis zu $b^{\frac{d(n-1)}{n}}$ Knoten [Knuth 1975].

Leider ist diese Annahme in vielen Spielen nicht angebracht, da die Gegner eine solche Allianz oft nicht eingehen und ein einzelner Gegner eine Verbesserung des Erwartungswerts für den suchenden Spieler sehr wohl in Betracht ziehen kann.

Der *paranoid*-Algorithmus schneidet für einen ziehenden Spieler einen Teilbaum B_t^j ab, wenn eine Verschlechterung seiner Position in diesem möglich ist. Es wird dabei nicht berücksichtigt, welche Auswirkung die Strategien des Teilbaumes für die anderen Spieler besitzen. Jegliche Möglichkeit von Kooperation, anonymer oder offener, wird von vornerein ausgeschlossen.

So wäre es denkbar, dass eine der möglichen Strategien sowohl für den ziehenden Spieler P_a als auch für den Folgespieler P_b eine Verbesserung ermöglicht. Fällt diese für P_b gar deutlicher als für P_a aus (anonyme Kooperation) oder beschließt P_b , die Verbesserung von P_a aus taktischen oder persönlichen Gründen in Kauf zu nehmen, so wird diese Strategie von P_b gewählt. Der *paranoid*-Algorithmus scheitert also dann, wenn er nicht berücksichtigt, dass strategische Kooperationen zu Gunsten eines ziehenden Spielers im Spiel integriert sind.

Im *paranoid*-Algorithmus werden an einem Knoten S_t genau dann falsche Abschneidungen durchgeführt, wenn ausser Acht gelassen wird, dass der bei S_t ziehende Spieler kein Interesse hat, den Erwartunsgwert des vorgehenden Spielers durch einen Zug a_i zu reduzieren, wenn dieser auch die Position des ziehenden gleichzeitig verschlechtert.

Dennoch lohnt es sich häufig, den Einsatz dieses Algorithmus zu berücksichtigen, da er eine wesentlich höhere Suchtiefe erreichen kann als der max^n-Algorithmus und somit Gefahren und Möglichkeiten in weiterer Zukunft besser identifizieren kann. Ein guter Vergleich und Analyse beider Ideen findet sich in [Sturtevant 2002]. Es wird darin auch gezeigt, dass die Wahl des Suchalgorithmus zum Teil von den Eigenschaften des untersuchten Spiels abhängt.

5.3 Lösung des MES über die Entropie-Nutzenfunktion $U_{H'}(\overrightarrow{e})$

Das MS-Entscheidungsproblem kann als Variante des Gefangenen-Dilemmas [Luce 1957] mit mehreren Gefangenen interpretiert werden. Dabei entsprechen die Entscheidungen der Mitspieler (besser: Mitinsassen) verdeckten Zügen.

Ähnlich wie in der ursprünglichen Version ist es zu gefährlich für einen Gefangenen, sich auf das Schweigen der anderen zu verlassen. Von diesem Gedanken getragen entscheiden sich womöglich alle, doch zu gestehen und verhindern so das Erreichen der für alle insgesamt besseren Situation, als wenn sie geschwiegen hätten. Das drei Gefangenen-Dilemma ist in Abbildung 5.3 dargestellt. Negative Bewertungen entsprechen dabei den Jahren Gefängnis, die einen Gefangenen

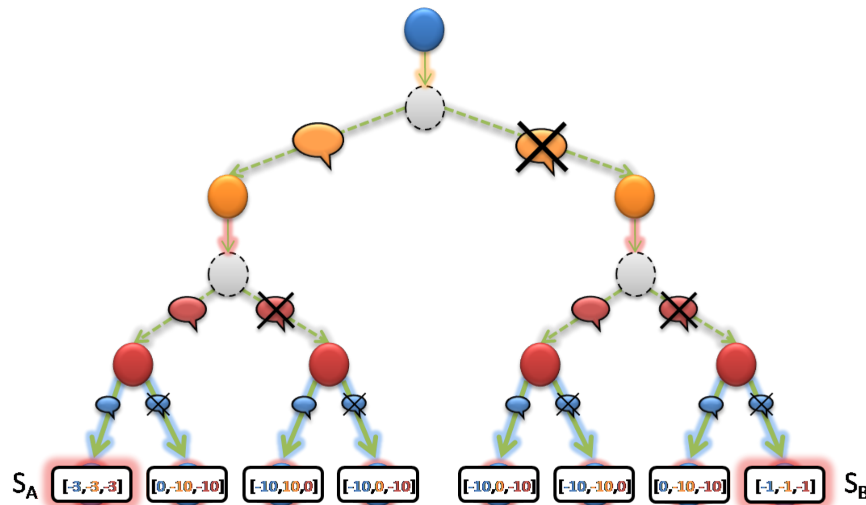


Abbildung 5.3: Eine Variante des Gefangenendilemmas für 3 Spieler. Aus der Befürchtung heraus, einer der Mitgefangenen könnte das Schweigen brechen, entscheiden sich alle dafür, die Tat zu gestehen und verpassen die gegenüber S_A bessere Situation S_B. Das Dilemma kann gelöst werden, wenn ein zusätzlicher Nutzen geschaffen wird. Beispielsweise veranlasst die Angst vor Racheakten die Gefangenen womöglich dazu, doch zu schweigen. Eine Kronzeugenregelung im Gegesatz begünstigt ein Geständnis.

bei der entsprechenden Konstellation erwarten.

Erst durch Hinzunahme weiterer Anreize und Bestrafungen bei den Spielerentscheidungen kann das scheinbare Paradoxon gelöst werden. Darunter fallen in der realen Welt beispielsweise Kronzeugenregelungen, neue Identitäten und das Schweigegesetz "Omertá" der Mafia.

5.3.1 Notwendigkeit Spieleabhängiger Nutzenfunktionen

Zusätzlich zu der Bewertung der erreichbaren Spielstellungen kann in Mehrspieler-Situationen eine weitere vergleichbare Größe, der Nutzen zur Lösung des Entscheidungsproblems eingeführt werden. Die Wahl zwischen gleichwertigen Stellungen beruht dann auf der unterschiedlichen Interpretation, wie sehr die Bewertungsvektoren dem eigenen Nutzen dienen.

Beim Nutzen handelt es sich um eine Spieler- und Spielabhängige Entscheidung, die vorgibt, mit welcher Herangehensweise der Spieler einen Sieg erreichen möchte (vgl. Kapitel 4.6 in [Rieck 2007]). Eine Funktion, die den eigenen Nutzen beschreibt, läßt sich gut anhand der zwei Beispiele aus Abbildung 5.4 erklären.

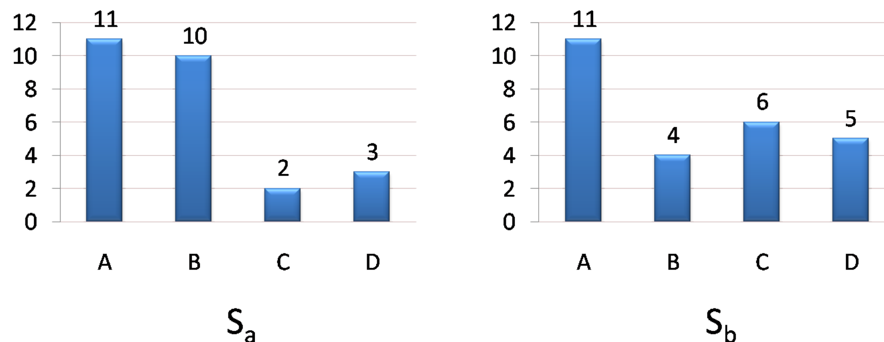


Abbildung 5.4: Der Nutzen für die eigene Strategie ermöglicht eine Entscheidung zwischen zwei gleichwertig erscheinenden Spielstellungen aus Sicht von Spieler A.

In beiden Fällen führt Spieler A und hat eine Bewertung von 11 erreicht. Seine Gegner kommen in der Summe auf eine Bewertung 15. Somit stimmen auch die Werte nach der Normierung der Vektoren für Spieler A überein ($\frac{11}{26}$). Möchte Spieler A die Anzahl der Spieler minimieren, die noch eine realistische Chance haben, ihm den Sieg streitig zu machen, so empfiehlt sich die Spielsituation S_a. Seine Nutzenfunktion versucht, Abstände zu einzelnen Gegnern zu maximieren. Tendiert Spieler A eher dazu, alle Gegner im Schnitt möglichst fern von ihm zu halten, scheint die Spielsituation S_b für diese Herangehensweise günstiger zu sein. In diesem Fall berücksichtigt die Nutzenfunktion auch die Varianz der Abstände der Gegner.

Die Herangehensweise kann auch deutlich vom gespielten Spiel abhängen. Entspricht die Bewertung von Spieler A beispielsweise den erreichten Siegpunkten im Spiel *Siedler von Catan*, so ist S_2 eindeutig zu bevorzugen. Spieler A befindet sich in einem 12-Punkte-Spiel unmittelbar vor dem Sieg und keiner seiner Gegner scheint in naher Zukunft bedrohlich zu werden. Die Chancen auf einen Sieg sind hier deutlich höher als in der ersten Spielsituation, wo mit Spieler B ein weiterer Spieler ebenfalls unmittelbar vor dem Sieg steht.

Genau umgekehrt sieht es aus, wenn die Bewertungen den Spielchips in einer Pokerrunde entsprechen. Die meisten menschlichen Spieler würden hier S_a für Spieler A favorisieren, da zwei der Gegner unmittelbar vor dem Ausscheiden stehen und somit die Chancen auf den Gesamtsieg deutlich höher stehen als in Spielstellung S_b. Hier zeichnet sich noch keine zuverlässige Tendenz ab und alle Gegner haben noch die Mittel, um den Sieg mitzuspielen.

Dieses Problem hat sich bei 2S-Spielen nicht gestellt, da es hier eindeutig war,

welche Herangehensweise zum Sieg führt: den positiven Abstand zum Gegner maximieren.

Eine Nutzenfunktion U erhält als Eingabe einen auf die Summe 1 normierten Erwartungswertvektor einer Stellung S_t und liefert einen gleichdimensionalen Vektor mit Werten zwischen -1 und 1 der Länge 1 zurück. Dabei deuten positive Werte auf einen Nutzen hin, während negative Werte dem Nutzen entgegenwirken. Die Sicht ist immer durch den suchenden Spieler P_s bestimmt, dessen Bewertung der Eingabevektor für U zugrunde liegt.

$$U(\|[J_S(S_t, P_1), \ldots, J_S(S_t, P_n)]\|) = \overrightarrow{u}$$

mit u_i als Nutzen der Erwartung von Stellung S_t aus Sicht des Spielers P_s.

Nutzenfunktionen spielen auch eine wichtige Rolle für Spiele mit stochastischen Elementen, wenn es zwischen Risiko und Sicherheit abzuwägen gilt. Sie haben ihren Ursprung in der Ökonomie und sind ein gutes Beispiel für die enge Verbindung zwischen den Wirschaftswissenschaften und der Spieltheorie. In der Ökonomie werden dabei üblicherweise Auswirkungen der Handlungen eines Unternehmens in Märkten mit mehreren Konkurrenten untersucht und bewertet. Es wundert daher nicht, dass John Nash 1994 für seine Forschungen auf dem Gebiet der Spieltheorie den Nobelpreis der Wirtschaftswissenschaften erhielt.

5.3.2 Die allgemeine Entropie-Nutzenfunktion $U_{H'}(\overrightarrow{e})$

Möchte man einen einzigen Suchalgorithmus für alle hier vorgestellten Spieleklassen entwickeln, so muss dieser eine Nutzenfunktion beinhalten, die für eine breite Menge an Spielen sinnvolle Herangehensweisen produziert. Idealerweise sollte die Nutzenfunktion an das jeweilige Spiel angepasst werden können, entweder durch von Hand einstellbare Parameter oder durch maschinelles Lernen.

Gemeinsam mit der AG-Spieltheorie an der Freien Universität Berlin wurde eine solche Funktion entwickelt: die Entropie-Nutzenfunktion U_{H^n}. Als Entropie H wird in der Informationstechnik der Erwartungswert des Informationsgehaltes eines Alphabets Z bezeichnet [Shannon 1963]. Dieser lässt sich mit der Formel

$$H(Z) = -\sum_{z \in Z} p_z * \log_2(p_z)$$

berechnen. Als p_z wird die Wahrscheinlichkeit des Aufkommen des Zeichens z aus Z angegeben. Mit $-\log_2(p_z)$ der Informationsgehalt des Zeichens z ("wie

aussagekräftig ist das Fehlen bzw. Vorhandensein des Zeichens z in einer Kette aus $Z*$").

Leicht uminterpretiert lässt sich H als ein Maß verstehen, wie sehr sich die Wahrscheinlichkeiten der Zeichen eines Alphabetes gleichen.

Die normierte Entropie H' mit

$$H'(Z) = \frac{H(Z)}{\sum_{z \in Z} \log_2(p_z)}$$

liefert Werte aus dem Intervall $(0, 1]$ zurück. Der Maximalwert 1 von H' wird genau dann erreicht, wenn alle Zeichen $z \in Z$ mit der gleichen Wahrscheinlichkeit auftreten. Kleinere Werte deuten auf ungleich verteilte Wahrscheinlichkeiten. Dies sind Eigenschaften, die für eine allgemeine Nutzenfunktion gut verwertet werden können. Denn:

- liegt ein Spieler P_i über dem Durchschnittswert μ_e des Bewertungsvektors $\overrightarrow{e}$ mit $\mu_e = \frac{\sum e_i}{|\overrightarrow{e}|}$, wird er vesuchen die Entropie H' der Bewertungen seiner Gegner zu minimieren, so dass alle gleich schlecht werden und sich damit der durchschnittliche Abstand zu ihnen vergrößert. Es gilt also den Wert

  $$U_{H'.\alpha}(\overrightarrow{e}, P_i) = H'(\overrightarrow{e} \setminus e_i) * (e_i - \mu_e)$$

  zu maximieren. Ein Entropiewert von 1 bedeutet dabei eine totale Kontrolle der Gegner und der Gesamtsituation. Dies kommt einer 2S-Situation gleich, bei der man gegen den Durchschnittswert der Gegner spielt. Jede Vergrößerung des Abstandes zu dem Durchschnittswert bedeutet bei einem Entropiewert von 1 definitiv eine eindeutige Verbesserung für P_i. Selbst wenn anzunehmen ist, dass sich in einigen Spielen Kooperationen der Gegner gegen den Führenden P_i dadurch bilden, dürfte diese Herangehensweise in den meisten Spielen sinnvoll sein.

- liegt ein Spieler P_i unter dem Durchschnittswert μ_e, so dürfte es in seinem Interesse sein, einerseits den Abstand zu μ_e zu minimieren und andererseits gleichzeitig die Entropie H' der gegnerischen Werte gering zu halten. Somit also

  $$U_{H'.\beta}(\overrightarrow{e}, P_i) = -H'(\overrightarrow{e} \setminus e_i) * (\mu_e - e_i)$$

  zu minimieren. Der zweite Fall $U_{H'.\beta}$ liefert auf alle Fälle negative Werte zurück, da beide Faktoren positiv sind und durch das Vorzeichen im

Produkt negativ werden. Im Gegensatz zur ersten Situation wird hier nun versucht, die Entropie zu reduzieren, da dies bedeutet, dass noch andere Spieler ebenfalls unter dem Durchschnitt liegen und eventuell als nächstes bald überholt werden können. Dies entspricht der Idee, dass man als Letzter versuchen sollte, immerhin noch den Vorletzten einzuholen und diese Regel rekursiv weiter anwenden kann, sobald man diesen eingeholt hat.

Offensichtlich gilt

$$U_{H'.\beta}(\overrightarrow{e}, P_i) = U_{H'.\alpha}(\overrightarrow{e}, P_i),$$

so dass sich die allgemeine Entropie-Nutzenfunktion U_{H^n}

$$U_{H'}(\overrightarrow{e}) = [u_{H'_1}, u_{H'_2}, ..., u_{H'_n}]$$

mit einer Zeile beschreiben lässt. Die einzelnen Komponenten des Ergebnissvektors

$$u_{H'_i} = H'(\overrightarrow{e} \setminus e_i) * (e_i - \mu_e)$$

orientieren sich an der eigenen Bewertung, dem Durchschnittswert μ_e des Bewertungsvektors (wenn normiert, dann ist μ_e immer $\frac{1}{|\overrightarrow{e}|}$) und der Ordnung der Gegner.

Der Term $H'(\overrightarrow{e} \setminus e_i)$ kann als Aussagekraft für das Abstandsmaß interpretiert werden. Ein Wert von 1 gibt an, dass der Abstand zum Durchschnitt der Gegner vollständig als Vorteil betrachtet werden kann und somit eine hohe Aussagekraft besitzt. Kleine Werte deuten an, dass die Aussagekraft reduziert ist, da nicht alle Gegner genau auf diesem Durchschnittswert liegen und zum Teil eine bessere Bewertung als diesen besitzen.

Einige Beispiele der Entropie-Nutzenfunktion sind in Abbildung 5.5 angeführt. Die Nutzenfunktion stellt keineswegs eine optimale Lösung für alle hier erfassten Spiele dar. Ein Computerspieler kann eine eigene Nutzenfunktion implementieren und diese in die allgemeine Suche einbinden. Nur in dem Falle, dass keine eigene Lösung vorhanden ist, sollte auf diese Funktion zurückgegriffen werden. Weitere Untersuchungen und Forschungen sollten unternommen werden, um die Entropie-Nutzenfunktion zu verfeinern und noch flexibler für die unterschiedlichen Spiele zu gestalten (siehe Abschnitt 7.2.3).

Damit die Ergebnisse einer Nutzenfunktion nicht noch um den Faktor $eval_{max}$

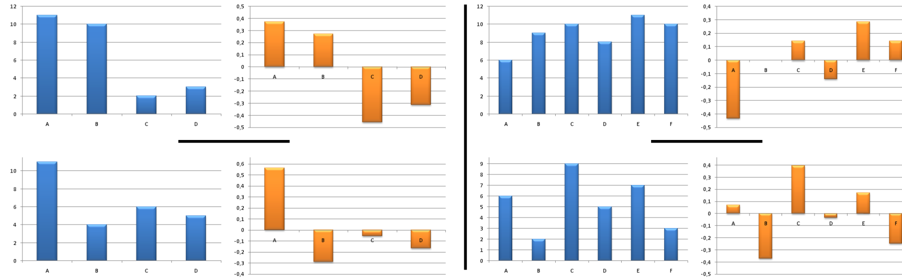


Abbildung 5.5: Einige Beispiele für die Ergebnisse der Nutzenfunktion $U_{H'}$. In Blau sind die Bewertungen für die einzelnen Spieler dargestellt; Orange das Ergebnis von $U_{H'}$. In der linken Bildhälfte sind zwei Beispiele für eine 4S-Spiel aufgeführt; rechts für ein 6S-Spiel.

multipliziert werden müssen, wird $eval_{max}$ einfach gleich 1 gesetzt. Somit liefern Stellungsbewertungen mit und ohne Verschachtelung innerhalb einer Nutzenfunktion Ergebnisse wie aus dem Abschnitt 3.1.1. Es bleibt den Entwicklern von Computerspielern überlassen, sich je nach Spiel und Konzept für oder gegen eine Nutzenfunktion zu entscheiden.

5.4 Der $UPP - max^n$-Suchalgorithmus

Der $UPP - max^n$-Algorithmus ($Utility$, $Paranoid$, $Probability - max^n$) ist ein Suchalgorithmus, der für eine breites Spektrum an Spielen konzipiert wurde. Er ist in der Lage, für sowohl deterministische 2S-Spiele mit perfekter Information als auch für komplexere MS-Spiele mit Mehrfachzügen und Zufallselementen eine annähernd optimale Lösung zu finden.

Er bietet die Möglichkeit, einzelne Komponenten leicht auszutauschen oder zu erweitern. Möchte man dies nicht, so beruht die Suche auf allgemein sinnvollen Verfahren. Darunter:

- Verwendung der Entropie-Nutzenfunktion $U_{H'}(\overrightarrow{e})$ zur Behebung des MS-Entscheidungsproblems. In 2S-Situationen liefert $U_{H'}(\overrightarrow{e})$ einfach den normierten Abstandsvektor beider Bewertungen zurück und kann somit auch für 2S-Spiele verwendet werden, ohne das Ergebnis zu verfälschen.

- Iterative Suche bis zu einer vorgegebenen Suchtiefe mit vorzeitigem Abbruch bei Zeitüberschreitung oder manuell ausgelöster Unterbrechungen. Der optimale Zug der zuletzt durchlaufenen Iteration wird dann zurückgeliefert.

- Spielersuchtiefe zur Handhabung von Mehrfachzügen. Dabei wird die Spielersuchtiefe dann dekrementiert, wenn es nach einem Zug zu einem Spielerwechsel kam. Dies hat keine Auswirkungen auf Spiele ohne Mehrfachzüge. Zusätzlich zur Spielersuchtiefe kann eine maximale Zugsuchtiefe eingestellt werden, um die Laufzeit des Algorithmus besser regeln zu können.

- Verwendung von Transpositionstabellen zur Beschleunigung der Suche. Stellungen, die erneut vorgefunden werden, können auf die bereits berechneten Zuglisten, Bewertungs- und Erwartungswerte zurückgreifen. Für Spiele, bei denen die Generierung eines eindeutigen Schlüssels für jede Stellung nicht realisiert ist, können Transpositionstabellen deaktiviert werden, indem für jede Stellung der ignorierte Schlüsselwert 0 zurückgeliefert wird.

- Auslagerung der Zuggenerierung zum Spieler. Dieser liefert eine eigens sortierte Zugliste mit den ihm zur Verfügung stehenden Informationen. Somit spielt es für den Suchalgorithmus keine Rolle, ob es sich um ein Spiel mit oder ohne vollständige(r) Information handelt. Zur Sortierung der Zugliste greift der Spieler auf den aktuellen Knoten der Suche und der Transpositionstabelle zurück. Somit kann er selbst entscheiden, welche Sortierungsheuristik angewendet wird. In *jGameAI* sind die Suchheuristiken $a_{i \rightarrow score} = Eval_S$ und $a_{i \rightarrow score} = J_S$ verallgemeinert implementiert und können auf Wunsch von Spielern verwendet werden.

- Automatische Erkennung von Zufallsknoten anhand der generierten Zuglisten. An diesen findet eine Zerlegung der optimalen Strategie in Teilstrategien statt, denen die Wahrscheinlichkeit des jeweiligen Zufallsereignisses zugewiesen wird. Als Ergebnise liefert der Algorithmus sowohl für deterministische als auch nicht-deterministische Spiele einen Strategiebaum zurück. Im ersten Fall besteht dieser dabei nur aus einer Entscheidungsvariante mit der Wahrscheinlichkeit von 1.

Der $UPP - max^n$-Algorithmus ist im *jGameAI* integriert und benötigt aus der Bibliothek nur die generischen Klassen eines Spielers, eines Zustandes, eines Spielzuges und einen Strategiebaum. Dies können die in jGameAI realisierten Klassen der implementierten Spiele sein oder eigene, die sich an die Schnittstellen der Bibliothek halten. Mehr Informationen und die offenen Programmcodedateien sind unter [jGameAI (URL)] zu finden.

5.4.1 *paranoid*-Pruning im $UPP - max^n$-Suchalgorithmus

Der große Vorteil des *paranoid*-Ansatzes ist es, viele Abschneidungen durchführen zu können, was wesentlich zur höheren Suchgeschwindigkeit beiträgt. Der potentielle Fehler, der aufgrund der möglicherweise fehlerhaften paranoiden Annahme entsteht, verhindert oft, dass das eigentliche Gleichgewicht des Suchbaumes gefunden wird [Sturtevant 2002].

Mit der Verwendung der Entropie-Nutzenfunktion, werden diese Fehler und somit auch ihre negativen Auswirkungen minimiert. Besitzen alle Gegner die gleiche Bewertung, so ist die paranoide Annahme gerechtfertigt und der Abstand zu den Gegnern kann als eigene Erwartung verwendet werden. In diesem Falle liefert die Entropie-Nutzenfunktion eben genau diesen Abstand, da hier die Aussagekraft gleich 1 ist.

Für weniger aussagekräftigere Ergebnisse reduziert sich die Wirkung des Abstands auf die Erwartung eines Spielers. In solchen Spielsituationen erreicht der suchende Spieler seltener sehr hohe oder sehr niedrige Erwartungswerte, die zu Abschneidungen führen könnten, da die Erwartung durch die Aussagekraft beschränkt ist. Dies reduziert die Wahrscheinlichkeit von paranoiden Abschneidungen an falsch interpretierten Spielstellungen.

Es bietet sich also an, wie im *paranoid*-Algorithmus Abschneidungen durchzuführen. Sollte ein Gegenspieler P_B den Nutzen des Vorgängers P_A reduzieren können, so verhindert P_A diese Situation, indem er seinen zu dieser Situation führenden Zug a_B nicht ausführt. Alle weiteren, noch nicht untersuchten Antwortzüge auf a_B des Gegners können abgeschnitten werden.

Unter den abgeschnittenen Zügen können sich nach wie vor ebenfalls Züge befinden, die unberechtigter Weise abgeschnitten werden. Beispielsweise solche die eine Verbesserung von beiden Spielern P_A und P_B bewirken. Mit der Entropie-Nutzenfunktion wird die Wahrscheinlichkeit eines Zuges reduziert, der zwei oder mehrere Spieler gleichzeitig verbessert, da die Bewertungsänderungen in Relation zu der Entwicklung der Gegner gestellt werden.

Daher schneidet $UPP - max^n$-Algorithmus seltener für das Gleichgewicht des Suchbaumes relevante Teilbäume ab und behebt somit eine der größten Schwächen des *paranoid*-Algorithmus.

Das *immediate pruning* kann weiterhin angewandt werden, da die Nutzenfunktion per Definition im Falle eines Sieges für Spieler P_i an der i. Stelle des zurückgelieferten Vektors den maximalen Bewertungswert 1 zurückliefert.

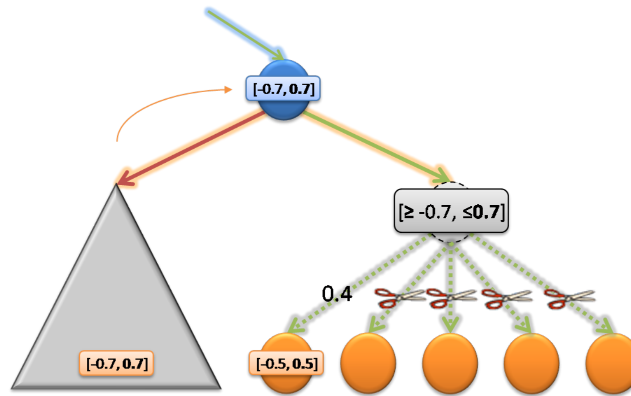


Abbildung 5.6: Abschneiden an einem Zufallsknoten: Spieler Orange kann Zufallsereignisse aus dem Suchbaum abschneiden, sobald klar wird, dass sie keine Bevorzugung des Zufallsknotens gegenüber einer bisherigen Bewertung mehr bewirken können.

5.4.2 Abschneidungen an Zufallsknoten in $UPP - max^n$

Das Abschneiden von Teilbäumen an einem Zufallsknoten birgt Gefahren. Da jedes noch so unwahrscheinliche Zufallsereignis dennoch eintreten kann und daher eine Berücksichtigung erfordert, möchte man auf alle Eventualitäten vorbereitet sein. Schneidet man an einem Zufallsknoten vorzeitig Zufallsereignisse ab, so trifft man diese Entscheidung aufgrund einer über die Wahrscheinlichkeiten der Folgestrategien gemittelten Erwartung und übersieht dabei eventuell die Risiken oder Möglichkeiten einzelner abgeschnittener Zufallsereignisse.

Wie lassen sich dennoch sinnvolle Abschneidungen durchführen? Dazu betrachten wir die Abbildung 5.6.

Die Berechnung des ersten Zuges ergab für Spieler Orange eine Bewertung von 0.7. Beim Betrachten des zweiten Zuges stellt der Spieler fest, dass er bereits nach dem ersten der möglichen Zufallsereignisse nicht mehr weiter zu rechnen braucht. Selbst wenn die verbleibenden vier Zufallsereignisse für Spieler Orange die maximale Bewertung 1 mit sich bringen, wird er im Schnitt nur den Wert

$$Eval_{Org}(S_t, P_{Org}) = 0.4 \cdot (-0.5) + (1 - 0.4) \cdot 1 = 0.4$$

erreichen können.

Sei $\hat{p}^j$ die summierte Wahrscheinlichkeit der ersten j Zufallsereignisse

$$\hat{p}^j = \sum_{i=1}^{j} p(a_i)$$

an einem Zufallsknoten S_t und e_i^jdes nach den ersten j Zufallsereignissen gemittelten Erwartungswertes für Spieler P_i zum Zustand S_t. Sei P_a der Spieler, der zum Zufallsknoten gezogen hat, und E_{P_a} sein bisher höchster erreichter Bewertungswert. So wird P_a bei dem Zufallsereigniss a_i abschneiden können, für das die Ungleichung

$$E_{P_a} > e_a^{(i-1)} + (1 - \hat{p}^{(i-1)})$$

zum ersten Mal falsch ist (vgl. [Russel/Norvig 2004]).

Es lohnt sich, für eine gute Beschleunigung die Zufallsereignisse ihrer Wahrscheinlichkeit nach zu sortieren, da man somit möglichst früh Abschneidungen vornehmen kann. In [Sturtevant 2003-1] wird zudem vorgeschlagen, Züge die zu Zufallsknoten führen, als letztes zu betrachten, da man hier das Abschneiden am letzten Teilbaum (*last branch pruning)* anwenden kann.

Hat ein Spieler bisher abgeschnitten, sobald klar wurde, dass er sich an einem Zufallsknoten nur noch verschlechtern kann, so erlaubt das *last branch pruning* auch Abschneidungen im Falle einer Verbesserung. Wird nach Betrachten der ersten j Zufallsereignisse klar, dass auch im schlechtesten Fall der Bewertung der restlichen Zufallsereignisse eine Verbesserung eintritt, so muss dieser Knoten nicht weiter berechnet werden und die restlichen Ereignisse können abgeschnitten werden.

Das *last branch pruning* kann genau dann eingesetzt werden, wenn die Eigenschaft

$$E_{P_a} < e_a^{(i-1)} + (1 - \hat{p}^{(i-1)}) \star (-1)$$

zum ersten Mal eintritt. Alle Zufallsereignisse mit einem Index größer-gleich i müssen dann nicht weiter berücksichtigt werden. Diese Methode lässt sich allerdings wirklich nur beim letzten Teilbaum eines Knotens einsetzen, da E_{P_a} bekannt sein muss. Es wird zudem empfohlen, diese Methode nur in der obersten Ebene einzusetzen, da man die Berechnung des strategischen Gleichgewichtswertes frühzeitig abbricht und nur eine Mindestschranke zurückliefert. Sollte

dieser Wert mit weiteren Erwartungswerten noch verglichen werden, kann es zu Verzerrungen und Fehlentscheidungen kommen; wenn beispielsweise alle abgeschnittenen Zufallsereignisse ebenfalls eine Verbesserung ermöglichen, ist E_{P_a} eigentlich höher als angegeben.

Beide Ansätze wurden im $UPP - max^n$-Algorithmus realisiert.

Kapitel 6

Lernen mit $TD - Prob^n(\lambda)$

Wie erlernt ein allgemeiner Computerspieler das Siegen? Viele Parameter und einstellbare Größen bieten sich für eine optimale Anpassung an das jeweilige Spiel an. Insgesamt umfasst die hier vorgestellte allgemeine rationale Suche:

- einen Suchbaum B_t^d mit einstellbarer Suchtiefe d
- eine $score_f$-Funktion als Vergleichsmaß für das Sortieren der Züge an einem Stellungsknoten
- eine Stellungsklassifizierung $C(S)$ mit n Stellungsklassen
- eine Menge mit k zur Auswahl stehender Funktionen der Stellungskriterien
- eine Bewertungsfunktion $Eval$ mit n Gewichtsvektoren $\overrightarrow{w_1}, \overrightarrow{w_2}, \ldots, \overrightarrow{w_n}$ mit jeweils k Elementen für die Gewichtung der Kriterien innerhalb aller n Stellungsklassen
- eine Auswahlfunktion arg_f für das Ermitteln der Erwartungswerte der Funktion J_S
- eine Nutzenfunktion $U(\overrightarrow{e})$ als Vergleichsmaß für zwei Erwartungswertvektoren

Je nach Spiel und Spieler wird sich eine unterschiedliche Konfiguration dieser Parameter anbieten. Initial könnten die meisten Parameter von Hand mit sinnvoll erscheinenden Werten und Funktionen belegt werden. Dennoch möchte man erlauben, dass diese Werte verändert werden können, um so eine optimale

Anpassung der Konfiguration an das Spiel zu ermöglichen. Hier kommen Erfahrung und Spielpraxis ins Spiel: aus der Beobachtung eigener und fremder Partien werden Rückschlüsse gezogen, so dass ein Computerspieler erkennt, in welche Richtung und wie stark er seine eigenen Parameter korrigieren muss.

Beispiel: Angenommen ein bestimmtes Kriteriums K_i wurde als zu schwach bewertet, obwohl dieses Kriterium für den Spielausgang sehr wichtig ist, so sollte durch maschinelles Lernen das Gewicht w_i nach und nach erhöht werden. Die fehlerhafte Bewertung des Kriteriums muss erkannt werden und mit einem Fehlermaß quantifiziert werden. Dies könnte online während einer gespielten Partie geschehen oder offline nach/vor einer gespielten Partie.

6.1 Maschinelles Lernen

Unter maschinelles Lernen fallen alle Verfahren, die versuchen, einem künstlichen System (auch Agent) durch Beispiele, Erfahrung und Training Erkenntnisse zu vermitteln, die dem System bei zukünftigen, noch unbekannten Situationen zu einer sinnvollen Beurteilung oder Reaktion verhelfen. Ein solches künstliches System stellt ein Computerspieler dar, dessen Parameter für die Suche innerhalb und die Bewertung von Spielsituationen erlernt werden können, um in zukünftigen Partien eine höhere Spielstärke zu erlangen. Dies kann durch eine akkuratere Kriterienwahl, korrigierte Gewichtsvektoren und eine bewährte Auswahlfunktion geschehen.

Man unterscheidet dabei allgemein bis zu drei Arten von maschinellem Lernen [Mitchell 1997]:

- Beim überwachten Lernen versucht das System, während einer Trainingsphase eine Überführung der Eingabedaten einer Trainingsmenge hin zu den gewünschten Ausgaben zu approximieren. Die Ausgabedaten werden dem System durch einen Trainer vorgelegt, der das System mit vorgegebenem Wissen trainiert. Für diese Anwendung eignen sich häufig neuronale Netzwerke, die mit Rückwärtspropagation lernen [Rojas 1993].

- Unüberwachtes Lernen: Hier liegt keine Zielmenge vor und das System muss versuchen, ohne direkte Anweisung die Trainingsmenge eigenständig zu klassifizieren. Diese Methoden kommen vor allem bei Clusteringverfahren und Datenreduktion mit niedrigem Informationsverlust zum Einsatz.

- Etwas gesondert ist die Methode des verstärkendes Lernens (*reinforcement*

learning). Sie wird zum Teil dem unüberwachten Lernen zugeordnet und stellt doch auch eine eigenständige Lernmethode dar. Nimmt man dem System auch noch die Eingabemenge, so muss es ohne Anleitung in seiner Umwelt agieren und durch eine Belohnungsfunktion eigenständig erkennen, welche Lern- und Korrekturschritte unternommen werden müssen. um ein bekanntes Ziel zu erreichen.

Idealisiert dargestellt, kombiniert maschinelles Lernen zum einen theoretisches Wissen und Erkenntnisse aus praktischen Erfahrungen. Das Wissen kann dabei anhand von Datenbanken, gespeicherten Partien, Schaltungen und festen Regeln in der KI verankert sein. Und zum anderen das intuitive Spielverständnis, das durch Training gewonnen werden kann und für zukünftige unbekannte Spielsituationen rationale Entscheidungen trifft.

Ziel von maschinellem Lernen ist es, eine Aktualisierungsanweisung für die zu erlernenden Paramter anzugeben, so dass erst einmal ein nicht näher benannter Fehler nach und nach minimiert werden kann. Dieser Fehler kann durch einen Trainer während dem überwachten Lernen angegeben werden oder in den weiteren Lernmethoden durch ein eigenes sinnvolles Fehlermaß ermittelt werden.

6.2 $TD-Prob^n(\lambda)$ zur Optimierung der Gewichtskoeffizienten

Die Beschränkung des Suchbaumes auf eine handhabbare Suchtiefe produziert unweigerlich Fehler, da die Folgen eines Zuges nicht vollständig untersucht werden. Zur Minimierung dieses Fehlers wird eine heuristische Bewertung an den Blattknoten durchgeführt und rekursiv zur Bestimmung des Erwartungswertes und stärksten Zuges zur Wurzel des Suchbaumes zurückpropagiert.

Maschinelles Lernen mittels temporaler Differenz versucht, diesen Fehler zu korrigieren, indem nach einer gespielten Partie der Spielausgang mit den berechneten Erwartungswerten des lernenen Computerspielers während der Partie verglichen wird. Es ist anzunehmen, dass die Erwartungswerte während einer Partie häufig nicht dem letztendlichen Spielausgang entsprechen.

Die Fehleinschätzungen erlauben Rückschlüsse über die Korrekturen der Gewichtskoeffizienten, da diese zu der Berechnung des Erwartungswertes beigetragen haben. Im Verlauf vieler gelernter Partien kann somit eine Optimierung der initial von Hand oder durch Zufall gesetzten Koeffizienten erreicht werden.

Basierend auf $TD - Leaf - ComplexEval(\lambda)$ dem erfolgreich angewandten Algorithmus aus [Block 2008] soll im Folgendem die Erweiterung $TD - Prob^n(\lambda)$ beschrieben werden, die das Erlernen von Gewichtskoeffizienten mittels temporaler Differenz in einem MS-Suchalgorithmus für nicht-deterministische Spiele realisiert. Der Einsatz wurde im $UPP - max^n$-Suchalgorithmus untersucht.

6.2.1 Konzept von $TD - Prob^n(\lambda)$

Der Lern-Algorithmus $TD - Prob^n(\lambda)$ soll die Anwendbarkeit von $TD - Leaf - ComplexEval(\lambda)$ auf probabilistische MS-Spiele ermöglichen. Dafür müssen einige Erweiterungen durchgeführt werden.

In Spielen wie *Siedler von Catan* und *Rommé* kann ein Spieler mehrfach hintereinander an der Reihe sein. Daher basiert die temporale Differenz von $TD - Prob^n(\lambda)$ nicht immer auf gleich weit voneinander entfernte, dem Spiele eigene Stellungen x_t und x_{t+1}. Eine dem lernenenden Spieler eigene Stellung x_t bezeichnet die t. Stellung, an dem dieser in einer Partie an der Reihe war.

Der Erwartungswert des Wurzelknotens eines Suchbaumes ist in probabilistische Spielen häufig eine gewichtete Summe von Erwartungen mehrerer Entscheidungsvarianten. Dies ist zum Beispiel genau dann der Fall, wenn die optimale Strategie über einen Zufallsknoten verläuft und in optimale Teilstrategien zerlegt wird (siehe Abschnitt 3.2.3). Jede Entscheidungsvariante einer Stellung x_t einer Partie besitzt einen eigenen Erwartungswert und damit eine eigene temporale Differenz zur Folgestellung x_{t+1}.

Die temporalen Differenzen aller Entscheidungsvarianten einer Strategie können durchaus unterschiedliche Vorzeichen haben und müssen daher unterschiedlich beim Lernen verwertet werden. Die Lernrate α muss in Abhängigkeit der Wahrscheinlichkeit einer Entscheidungsvariante gewählt werden. Da die Summen der Wahrscheinlichkeiten aller Entscheidungsvarianten einer Strategie stets 1 ergibt, erscheint es logisch, als Lernrate für eine Variante v_t^i den Wert $\alpha \cdot p_{v_t^i}$ zu verwenden.

Die Klassifizierung von Stellungen erfordert zudem, dass der Lernalgorithmus für eine Stellung x_t nur die Gewichtsvektoren korrigiert, die von den Entscheidungsvarianten zur Berechnung der Erwartungswerte verwendet wurden.

6.2.2 Optimieren der Gewichtskoeffizienten

Die verallgemeinerte Berechnung der temporalen Differenz

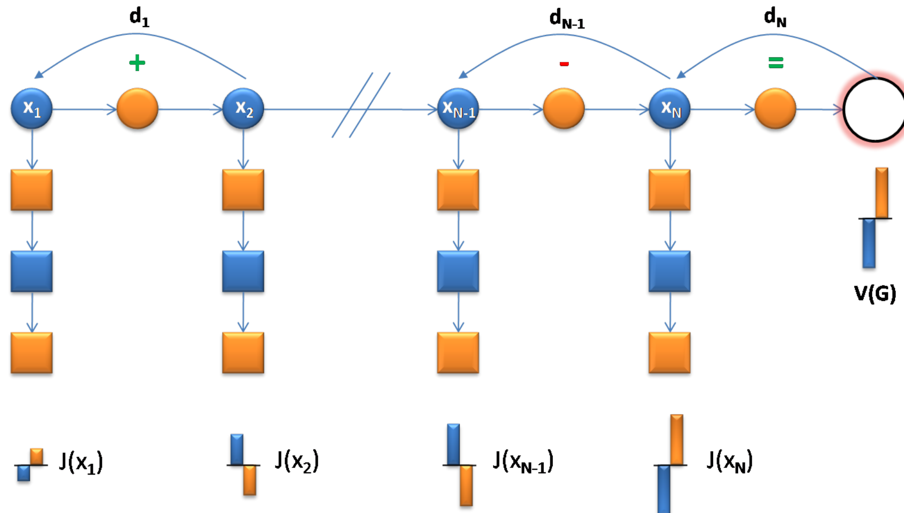


Abbildung 6.1: Schematische Darstellung temporaler Differenzen in einem 2S-Spiel. Die TD d_t zum Zeitpunkt t entspricht der Differenz zum Erwartungswert der nachfolgenden eigenen Stellung $(J(x_{t+1}) - J(x_t))$. Für den letzten Knoten wird die Differenz zum Spielausgang $v(G)$ verwendet. Spieler Blau ist in diesem Beispiel der lernende Spieler.

$$d_t := J(x_{t+1}) - J(x_t)$$

[Sutton 1990], welche die Differenz der Erwartungswerte zweier aufeinander folgender eigener Stellungen ermittelt, lässt sich in

$$d_t := J_v(x_{t+1}, v_{t+1}) - J_v(x_t, v_t)$$

überführen, wenn v_t die einzige Entscheidungsvariante ist, auf welcher der Erwartungswert der Stellung x_t basiert und die Funktion J_v dem Erwartungswert der übergebenen Entscheidungsvariante zurückliefert.

Die allgemeine Formel für die Optimierung eins Gewichtsvektors w mittels Temporaler Differenz lautet dann

$$w := w + \alpha \sum_{t=1}^{N} \nabla J_v(x_t, v_t) \cdot \left[\sum_{j=t}^{N} \lambda^{j-t} d_j \right]$$

(vgl. [Baxter 1997]).

Dabei wird für jede dem Lernenden eigene Stellung x_t die temporale Differenz zum Zeitpunkt t und die aller restlichen Folgestellungen der Partie ("Echo") mit

$$\sum_{j=t}^{N} \lambda^{j-t} d_j$$

berechnet. Der Term gibt die Größenordnung der Korrekturen an, die zum Zeitpunkt t durchgeführt werden müssen. Die Propagationsstärke λ, mit $0 \leq \lambda \leq 1$, regelt den Einfluss weiter entfernter Fehleinschätzungen bei der Anpassung der Gewichte, die zur Bewertung der Entscheidungsvariante v_t der Stellung x_t verwendet wurden. Die Zahl N enstspricht dabei der Anzahl aller dem lernenden Spieler eigenen Stellungen einer gespielten Partie ($N < |\omega|$).

Die Richtung der Korrekturen erhält man über den Gradienten

$$\nabla J_v(x_t, v_t)$$

der Entscheidungsvariante, der bei Stellung x_t gefundenen optimalen Strategie. Der Vektor entspricht dabei dem des Gradienten des Gewichtsvektors w_k, mit $k = C'(v_t)$.

Über die Lernrate α kann eine Konvergenz des Lernvorgangs herbeigeführt werden. Die Wahl einer von der Stellungsklasse der Entscheidungsvariante abhängigen Lernrate $a_{C(S_t)}$ ermöglicht differenzierteres Lernen. Daraus ergab sich die letztendlich in [Block 2008] verwendete Formel

$$w_k = w_k + \alpha_k \sum_{t=1}^{N} \nabla J_v(x_t, v_t) \cdot \left[\sum_{j=t}^{N} \lambda^{j-t} d_j \right]$$

für die Berechnung der Aktualisierungsschritte aller verwendeten Gewichtsvektoren. Dabei werden für die Anpassung eines Gewichtsvektors w_k nur diejenigen Stellungen berücksichtigt, deren Erwartungswert auf die Bewertung einer Stellung S_t zurückzuführen ist, für die $k = C(S_t)$ gilt.

Die wesentliche Erweiterung von $TD - Prob^n(\lambda)$ besteht darin, auch aus Strategien zu lernen, die auf mehrere Entscheidungsvarianten zurückzuführen sind. Dies bedeuetet, dass für eine Stellung x_t die Variable V_t aus mehreren v_t besteht (siehe Abschnitt 4.4.3). Der für eine konkrete Entscheidungsvariante v_t^i berechnete Optimierungsschritt muss mit der Wahrscheinlichkeit $p_{v_t^i}$ multi-

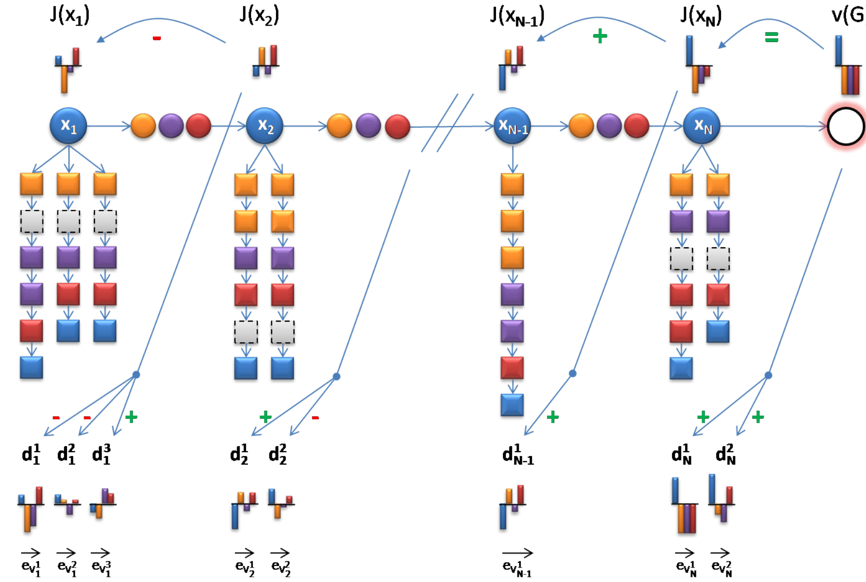


Abbildung 6.2: Schematische Darstellung temporaler Differenzen in einem MS-Spiel. Die TD d_t^i zum Zeitpunkt t entspricht der Differenz zwischen dem Erwartungswert $\overrightarrow{e_{v_t^i}}$ der Entscheidungsvariante v_t^i und dem Erwartungswert der nachfolgenden eigenen Stellung $(J(x_{t+1}) - J_v(x_t, v_t^i))$. Für den letzten Knoten wird die Differenz zum Spielausgang $v(G)$ verwendet. Spieler Blau ist in diesem Beispiel der lernende Spieler.

pliziert werden, da v_t^i mit eben diesem Anteil zu dem insgesamt fehlerhaften Erwartungswert der Stellung x_t beigetragen hat und daher nur in dieser Größenordnung zur Korrektur beitragen darf.

Des weiteren muss eine neue Berechnung der temporalen Differenz eingeführt werden. Unterschiedliche Entscheidungsvarianten der gleichen Strategie können unterschiedliche temporale Differenzen zur Folgestellung besitzen. Deswegen muss eine genauere Unterscheidung der temporalen Differenzen gefunden werden.

Die Formel

$$d_t^i := J(x_{t+1}) - J_v(x_t, v_t^i)$$

ermittelt die temporale Differenz zwischen einer einzelnen Entscheidungsvariante v_t^i und dem Erwartunsgwert der Folgestellung x_{t+1} (siehe Abb. 6.2).

Somit ergibt sich für alle Stellungsvarianten v_t^i, die über die Stellungsklasse k bewertet wurden, also $k = C'(v_t^i)$, der Aktualisierungsschritt

$$\omega_k := \omega_k + \alpha_k \cdot \sum_t^N \sum_i^{V_t} p_{v_t^i} \cdot \nabla J_v(x_t, v_t^i) \cdot \left[\sum_{j=t+1}^{N} \lambda^{j-t} d_j^i \right]$$

für jeden vom Lernenden verwendeten Gewichtsvektor w_k.

Für die Berechnung der temporalen Differenzen gelten die Bedingungen wie in [Block 2004]: positive Differenzen einer Stellung x_t werden nur genau dann gelernt, wenn die Vorhersage einer Entscheidungsvariante exakt eingetroffen ist zwischen den eigenen Stellungen x_t und x_{t+1}, da sonst mögliche Fehlzüge des Gegners gelernt werden könnten. Positive Differenzen, für die diese Bedingung nicht erfüllt ist, werden auf 0 gesetzt und besitzen somit keine Wirkung auf das Lernverhalten.

Negative Differenzen bedeuten auch im Falle eines Nicht-Eintretens der Vorhersage eine Fehleinschätzung der Situation und werden somit auf alle Fälle gelernt.

6.2.3 Diskussion und Ergebnisse

Die temporalen Differenzen nachfolgender eigener Stellungen $x_{t+2,t+3...}$ beeinflussen, wenn auch mit immer schwächerer Wirkung, die Korrekturen der Gewichtskoeffizienten zum Zeitpunkt x_t. Der abfallende Einfluss wird über die Propagationsstärke λ sichergestellt, welche die temporalen Differenzen $d_{t+2}, d_{t+3} \ldots$ mit den jeweiligen immer kleiner werdenden Faktoren $\lambda^2, \lambda^3 \ldots$ versieht (da $0 \leq \lambda \leq 1$). Dies geschieht innerhalb des Terms

$$\left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right]$$

während eines Aktualisierungsschrittes.

Der Ansatz ist für Spiele korrekt, an denen zwei aufeinander folgende eigene Stellungen x_t und x_{t+1} immer gleich viele Züge Abstand zueinander besitzen und somit nachfolgende Stellungen im gleichen Verhältnis mit berücksichtigt werden können.

In den hier behandelten Spielen ist die Entfernung zwischen zwei eigenen Stellungen nicht mehr zwingend konstant (z.B. Mehrfachzüge im Spiel *Siedler von Catan*). Die Differenzen nachfolgender Stellungen werden aber mit den

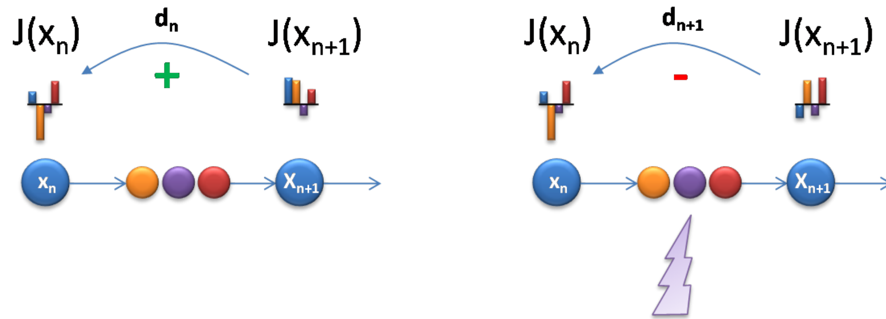


Abbildung 6.3: Erst durch die Wahl eines schlechteren Zuges (Fehlzug) durch Spieler violett entsteht eine negative temporale Differenz aus Sicht des Spielers blau, da er sich dadurch unerwartet verschlechtert hat.

gleichen Faktoren berücksichtigt. Hier müssen noch weitere Forschungen durchgeführt werden, um eine flexiblere Handhabung zu ermöglichen.

Bei positiven temporalen Differenzen dürfen nur dann Optimierungen durchführt werden, wenn die Züge der Entscheidungsvariante real ausgespielt wurden. Ansonsten besteht die Gefahr, dass die Fehler des Gegners erlernt werden, da sie irrtümlich für eine richtige Bewertung gehalten werden.

In MS-Spielen können auch im Falle negativer temporaler Differenzen Fehler der Gegner erlernt werden (Siehe Abb. 6.3). So besteht in einem vier-Spieler-Spiel die Möglichkeit, dass einer der Gegner zwischen zwei eigenen Stellungen einen Fehlzug durchführt. Womöglich entsteht dadurch eine negative TD, da dem Folgespieler ermöglicht wird, eine aus Sicht des Lernenden schlechtere Stellung zu bewirken, wenn dies zu seinem eigenen Vorteil geschieht.

Daraus resultiert eine Korrektur der Gewichtskoeffizienten, obwohl, die ursprüngliche Einschätzung richtig gewesen sein kann und somit zu einer Verstärkung der Gewichte führen müsste. Dies lässt vermuten, dass in MS-Spielen negative TD nur dann gelernt werden dürfen, wenn alle zwischenzeitlich gespielten Zügen denen der Entscheidungsvariante entsprechen.

Eine genauere Untersuchung, inwieweit dieser Fehler durch eine Nutzenfunktion und die paranoide Annahme, dass der gegnerische "Fehlzug" mit der Absicht geschah, die eigene Position zu schwächen und somit kein Fehlzug war, liegt noch nicht vor.

Zusätzlich muss noch untersucht werden, welche Erkenntnisse Entscheidungsvarianten ermöglichen, die eine positive temporale Differenz besitzen.

Kapitel 7

Beiträge und zukünftige Arbeiten

Die Entwicklung eines universalen Computerspielers, der in seiner Spiel- und Lernweise den Menschen ähnelt, ist noch nicht abgeschlossen. Viele weitere Punkte müssen noch geklärt werden. Mit dieser Arbeit wurde ein prinzipieller Ansatz vorgestellt, diesem Ziel näher zu kommen. Dabei wurde ein suchbaumbasierter Lösungsweg vorgeschlagen, der durch Abschneiden irrelevanter Teilbäume und Betrachten wichtiger Züge zuerst versucht, auch mit komplexeren Spielen zurecht zu kommen.

Die Entwicklung von Computerspielern hängt in dem hier vorgestellten Kontext stark von der Wahl der Stellungskriterien ab. Diese sollten wenn möglich von Meisterspielern erkannt und beschrieben werden. Es ist daher nicht Ziel dieser Arbeit gewesen, starke Computerspieler zu konzipieren. Viele eher wird Anwendern mit wenig Kenntnissen der Informatik oder geringem spieltheoretischen Wissen die Möglichkeit geschaffen, ihre intelligenten Konzepte in einen Rahmen leicht einzubinden, ohne sich mit vielen für die KI irrelevanten Problemen beschäftigen zu müssen.

7.1 Beiträge der Arbeit

Eine Klassifizierung von Spielen anhand ihrer Komplexität, Spieleigenschaften und allgemeiner Lösungswege wurde vorgestellt. Dabei wurden auch die Größe des Suchraumes und die Möglichkeiten in diesem nach besten Zügen zu suchen

berücksichtigt.

Mit der in dieser Arbeit eingeführten Notation zur formalen Beschreibung und Klassifizierung von Spielen, Such- und Lernalgorithmen konnte die Funktionsweise der Algorithmen $UPP - max^n$ und $TD - Prob^n(\lambda)$ erläutert werden. Dabei handelt es sich um allgemeine Such- und Lernalgorithmen, die für eine breite Menge von Spielen anwendbar sind. Die Grundgedanken beider Verfahren sowie ihre Stärken und Schwächen wurden diskutiert.

Die hier vorgestellten Algorithmen $UPP - max^n$ und $TD - Prob^n(\lambda)$ zur Modellierung eines allgemeinen Computerspielers wurden im Spieleframework *jGameAI* realisiert (siehe Kapitel 8).

7.2 Ausblick auf zukünftige Arbeiten

Der hier präsentierte Suchalgorithmus erlaubt noch viele Erweiterungen, die eine noch bessere Anpassung an das jeweilige Spiel ermöglichen. Dabei sollen vor allem die erfolgreichen Erweiterungen aus der Schachprogrammierung verallgemeinert in den Suchalgorithmus eingebunden werden. Dennoch sollen die Erweiterungen Vorteile für möglichst viele Spielen bewirken.

7.2.1 Zusätzliche Erweiterungen in $UPP - max^n$ einbauen

Suchbaumbasierte Verfahren, wie der $UPP - max^n - Algorithmus$, unterliegen immer der Gefahr des Horizonteffekts [Russel/Norvig 2004]. Dieser tritt ein, wenn ein Computerspieler eine Fehleinschätzung der aktuellen Lage trifft, da die benötigten Erkenntnisse weiter in der Zukunft liegen als die aktuelle Suchtiefe. Noch deutlicher tritt dieses Problem auf, wenn der tiefste zukünftige Zug beispielsweise ein Schlagzug bei Spielen wie Dame und Schach war. Wäre der Gegenspieler unmittelbar in der Lage, den Verlust wieder auszugleichen, so wird dies nicht erkannt, da die Suche an dieser Stelle abbricht.

Im Schachprogramm *FUSc#* [Block 2005] wurde das Problem mit einer Ruhesuche vermieden, die eine Suche nur an Knoten abbricht, die keine Schlag- oder Schachzüge enthielten. Leider bieten nicht alle Spiele eine so eindeutige Einteilung von Zugtypen, anhand derer man eine Ruhesuche definieren könnte. Eine Klassifizierung von Spielzügen wäre allerdings auch für weitere Erweiterungen wie die der Killerzüge [Steinwender 1995] interessant.

Allgemein könnte sich die Suche auf nur noch interessante und spielentscheidende Züge reduzieren. Dies kommt dem menschlichen Vorgehen wesent-

lich näher als das systematische Durchsuchen aller Möglichkeiten. Je nach der zur Verfügung stehenden Zeit würden dann nur die wichtigsten Züge untersucht werden. Die Klassifizierung könnte dabei maschinell erlernt und in gespielten Partien überprüft werden.

7.2.2 Stellungskriterien selbstständig entdecken

Beim Erlernen neuer Spiele bekommen Menschen häufig direkte Hinweise von erfahrenen Spielern, worauf im Spiel besonders zu achten ist. Auch leiten sich oft von den Spielregeln Wissen und Bedingungen ab, welche für siegreiches Spielen verwertet werden können. Die beiden "Starthilfen" werden auch einem Computerspieler in Form von Stellungskriterien und einer initialen Gewichtung mitgegeben.

Kaum ein menschlicher Spieler wird also bei Null anfangen müssen, wenn es darum geht, eigene siegreiche Strategien zu entwickeln. Im Verlauf vieler gespielter Partien beginnt der Spieler, aus der Erfahrung weiter führendes Wissen abzuleiten. Eine Neugewichtung des bisherigen Wissens erfolgt. Der Schritt wird bei Computerspielern durch das maschinelle Lernen gut abgedeckt und ist Gegenstand der vorliegenden Arbeit.

Damit ist allerdings der womöglich wichtigste Schritt im Erlernen eines Spieles durch menschliche Spieler nicht abgedeckt: das eigenständige Entdecken neuer Stellungskriterien. Kein Mentor kann einem Spieler alle denkbaren Stellungskriterien beibringen. Die Einstellung und die Intuition beider mögen sich auch deutlich von einander unterscheiden, so dass sie unterschiedliche Bewertungen der Kritierien vornehmen. Eine eindeutige Menge für siegreiches Spielen liegt womöglich gar nicht vor.

Wie finden Menschen eigenständig neue Stellungskriterien? Diesen Prozess zu analysieren und in maschinellen Lernalgorithmen zu formalisieren, würde die Entwicklung von Computerspieler nun vollständig vom Mentor Mensch befreien. Ein Spiel würde dann wirklich eigenständig vom Computer erlernt. Davon könnten auch die Menschen profitieren, die eventuell somit neue wichtige Aspekte von Spielen aufdecken könnten.

Bei diesem Problem handelt es sich allgemein um die Suche nach siegesträchtigen Mustern im Raum aller möglichen Muster, die im Verlauf eines Spieles auftreten können. Gesucht werden alle Muster und ihre Submuster, die für einen Sieg relevant erscheinen. Jedes solcher Muster kann anschließend mit einem einzelnen Stellungskriterium auf Vorkommen untersucht werden, die Gewichtung

der Stellungskriterien anschließend mit den vorgestellten Lernmethoden angepasst werden.

Zwei unterschiedliche Ansätze erscheinen dafür denkbar:

- Ableiten von Siegmuster aus den Spielregeln: In den Spielregeln sind fest die Bedingungen für Sieg und Niederlage verankert und beschrieben. Die Bedingungen treten häufig nur bei bestimmten Mustern im Spiel auf (etwa Mattmuster für Schach oder Vierer-Reihen bei 4-gewinnt!). Im Raum aller denkbaren Muster (eine wahrscheinlich unendliche Menge) liegen somit zumindest für einige Muster Angaben, dass sie wichtig für einen Sieg sind. Diese könnte man mit einem maximalen Gewicht versehen. Die genaue Anpassung der Gewichte an Submuster erfolgt dann durch Training. Für ein solches Vorgehen müssten die Spielregeln in einer formalen Sprache vorliegen, aus denen sich die Muster ableiten ließen. Auch ist nicht unbedingt klar, wie genau die Ableitung formalisiert werden kann.

- Evolutionäres Durchsetzen zufällig generierter Muster: Mit einem Satz an zufällig generierten Mustern ausgestattet, tritt ein Computerspieler in Trainingspartien gegen andere auch noch unerfahrene Gegner an. Durch evolutionäres Selektieren und "sinnvolles" Mutieren der Muster von den siegreichen Computerspielern könnten sich dabei einige Erfolg versprechende durchsetzen. Auch könnte man das Trainieren mit starken Computerspielern durchführen; mit der Gefahr, dass letztendlich nur deren Muster nach und nach kopiert werden. Bedenkt man die unglaubliche Anzahl der möglichen Muster in einem Spiel und die verschwindend geringe Anzahl an Siegesmustern, so ist es gut möglich, dass dieser Ansatz scheitert. Es käme einer Suche im Heuhaufen gleich, die noch nicht weiß, dass sie einer Nadel gilt.

Während der erste Ansatz ein nummerischer ist und wahrscheinlich eher auf dem Gebiet der Mathematik eine Lösung finden wird, dürfte die zweite Idee in der Informatik viel Aufmerksamkeit finden. Eine allgemein gute praktische Lösung hierfür könnte vielen ähnlichen Problemen zu neuen Lösungswegen verhelfen.

7.2.3 Nicht-lineares Entfernungsmaß bei $U_{H'}$

Die Entropie-Nutzenfunktion $U_{H'}$ besitzt eine lineare Entfernungsfunktion zum Mittelwert der Bewertungen der Gegner, anhand derer die eigene Bewertung ins

Verhältnis gebracht wird $(e_i - \mu_e)$. Dadurch wird ausgedrückt, dass beispielsweise eine doppelte Entfernung zum Mittelwert die Wahrscheinlichkeit eines Sieges je nach Vorzeichen verdoppeln oder halbieren kann.

Das entspricht in vielen Spielen nicht unbedingt den realen Gegebenheiten. Ein Spieler mit nicht linearem Entfernungsmaß bietet sich als Lösung an.

Ein starker MS-Algorithmus muss auch diesen Punkt berücksichtigen. Eine zukünftige Entwicklung von jGameAI besteht in der Integration eines maschinell lernbaren nicht-linearen Entfernungsmaßes $d : [-1 \ldots 1] \times \mu_e \rightarrow [-1 \ldots 1]$, der Entropie-Nutzenfunktion $U_{H'}$. Es muss möglich sein, für jedes Spiel und seine Eigenarten eine eigenständige Nutzenfunktion zu entwickeln.

$$u_{H'_i} = H'(\overrightarrow{e} \setminus e_i) \cdot d(e_i, \mu_e)$$

Inwiefern die Entropie-Nutzenfunktion sich als Grundlage für dieses Vorhaben eignet, muss noch untersucht werden. Des Weiteren muss geklärt werden, aus welchen Erkenntnissen eines Spielverlaufes welche Änderungen am Entfernungsmaß vorgenommen werden können.

7.2.4 Risiko in Probabilistischen Spielen

Durch den Verlust der vollständigen Information und durch die unmögliche exakte Vorhersage entstehen für die Suche neue Probleme, die nicht mehr mit eindeutigen Lösungswegen behoben werden können.

Wie geht man zum Beispiel mit dem Wissen um, dass als Folge eines Zufallsereignisses eine eigene Niederlage erzwungen werden kann? Drastisch formuliert wird diese Problematik anhand des folgenden Beispiels eines (zugegeben etwas martialisch anmutenden) Gewinnspiels:

> Entscheidet sich ein Teilnehmer des Gewinnspiels für die linke Tür, so erwartet ihn der sichere Gewinn von 100 Euro (u_1). Wählt er die rechte Tür, so gewinnt er zu 99% (p_2) eine Million Euro (u_2). Im Falle der verbleibenden restlichen 1% (p_3) trifft er allerdings auf einen hungrigen frei herumlaufenden Löwen (u_3). Wie sollte sich ein optimaler Spieler entscheiden?

Solche Beispiele führen vor, dass in Spielen mit Zufallsereignissen und unvollständiger Information keine eindeutige optimale Strategie mehr im bisherigen Sinne vorliegt. Eine Entscheidung in Abhängigkeit aller Wahrscheinlichkeiten

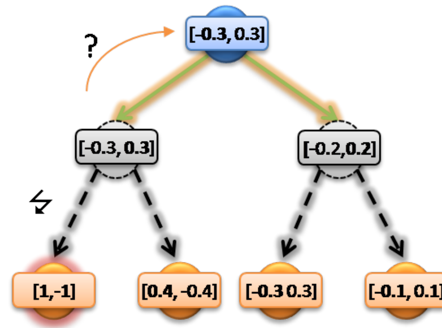


Abbildung 7.1: Risikoreiche Zufallsereignisse müssen identifiziert werden.

erscheint sinnlos, wenn die möglichen Konsequenzen von Zufallsereignissen und Zügen (u_i) schwer miteinander zu vergleichen sind.

Zwar versucht die Bewertungsfunktion terminale und nicht-terminale Zustände miteinander vergleichbar zu machen, dennoch entspricht eine Niederlage eindeutig einer anderen Situation als eine schlecht bewertete Stellung. Erstere ist unwiderruflich und kann nicht mit einem starken Antwortzug eventuell noch gekippt werden.

Wenn in einem über ein Zufallsereignis erreichbaren Teilbaum eine Niederlage für den ziehenden Spieler vorkommen sollte, so handelt es sich bei diesem Zufallsereignis um ein Risiko. Abbildung 7.1 verdeutlicht diese Problematik und zeigt, dass das Mitteln der Erwartungswerte das Risko verdeckt und somit die Möglichkeit einer sofortigen Niederlage birgt.

Die noch größere Gefahr bei Zufallsknoten entsteht dann, wenn man an diesen Abschneidungen durchführt. Es besteht die Möglichkeit, dass eine mögliche terminale Stellung und somit ein eigener Sieg oder eine eigene Niederlage übersehen werden. Dies kann nur durch das Untersuchen aller Teilbäume verhindert werden und führt zum Verlust des Beschleunigungseffektes eines Abschneidens.

Eine sich anbietende triviale Heuristik untersucht, in wie vielen Zügen eine terminale Stellung frühestens möglich ist ($depth_{min}$) und liefert für eine Stellung S_{t+j} im Suchbaum B_t^d den Wert $\frac{d-j}{depth_{min}}$ zurück. Erreicht diese Wahrscheinlichkeit einen lernbaren Schwellwert ϵ_p, wird ein Abschneiden dieses Teilknotens verboten.

Dies eignet sich gut für Spiele mit einer festen für den Sieg zu erreichenden Punktezahl. Eine ähnliche Heuristik wird in [Thomas 2003] im Spiel *Siedler von Catan* verwendet, um die Siegesaussichten von Stellungen zu bewerten. Sie stellt

fest, in wie vielen Runden ein Spieler frühestens gewinnen kann, wenn die Würfel immer am günstigsten fallen und der Spieler somit alle zum Sieg noch fehlenden Bau- und Kaufvorhaben realisieren kann.

Kapitel 8

Das Spieleframework jGameAI

Das Projekt *jGameAI* ist ein in der Programmiersprache *Java* entwickeltes Framework, mit dessen Hilfe man eigene Computerspieler für unterschiedliche Spiele realisieren kann. Die erfassten Spiele decken ein breites Spektrum ab: von 2-Spieler-Spielen mit vollständiger Information wie Schach und Dame bis hin zu komplexeren Mehrspieler-Spielen mit unvollständiger Information und Zufallselementen, etwa bei Poker und Siedler von Catan.

Im Framework ist ein verallgemeinerter Suchalgorithmus eingebunden, der für ein breites Spektrum an Spielen die Suche übernimmt. Dafür wird ein Suchbaum zukünftig erreichbarer Stellungen erstellt und anhand des wahrscheinlichsten Spielverlaufs die jeweils besten Züge für jede Stellung ermittelt.

Als eigene Leistung liefern Entwickler von Computerspieler nur noch eine Bewertungsfunktion, die von einem verallgemeinerten Suchalgorithmus verwendet wird. Mit Bewertungsfunktionen lassen sich zwei unterschiedliche Stellungen vergleichen und somit diejenige ermitteln, die für einen ziehenden Spieler günstiger erscheint. Diese können auf angebotene Lernmethoden zurückgreifen und so die Intelligenz des Spielers optimal an das Spiel anpassen.

Ein allgemeiner Lernalgorithmus wurde entwickelt um für das gesamte Spektrum an erfassten Spielen die Anpassung von Koeffizienten mittels maschinellem Lernen zu übernehmen. Das Lernen kann anhand einer generischen graphischen Oberfläche analysiert und beobachtet werden. Ebenfalls können weitere Parameter der Such- und Sortierheuristiken maschinell erlernt und vom Computer-

spieler mitgebracht werden.

Der direkte Vergleich zweier oder mehrerer Computerspieler findet dadurch nur auf der Ebene der rationalen Suche statt und wird nicht verzerrt durch unterschiedliche Rechenleistung, Suchalgorithmen, Programmiersprachen und Implementierung. Dadurch wird die Intelligenz des Spieler isoliert und von der technischen Realisierung getrennt.

8.1 Motivation und Zielsetzung

Die Idee für das Framework entstand aus dem Wunsch heraus, die gewonnenen Erkenntnisse des *FUSc#*-Schachprojekts der Freie Universität Berlin auch für weitere Spiele verwenden zu können [Block 2005]. Idealerweise sollte dabei ein Computerspieler entstehen, der in der Lage ist, sämtliche Spiele zu erlernen, die sich formal mit Zuständen und Übergangsfunktionen beschreiben lassen.

Ähnliche Forschungen, die einem universalen Spieler gelten, existieren bereits weltweit. So betreibt die Universität Stanford in Kalifornien, das *general game playing* Projekt [Standford (URL)], mit dessen Hilfe Spieler für beliebige Spiele realisiert werden, die sich in der eigens entwickelten Beschreibungssprache *game description language (GDL)* formalisieren lassen [GDL (URL)]. Bisher werden damit allerdings nur deterministische Spiele erfasst.

Das Zusammenführen von Lösungen für Spiele liegt nahe, da sich die Ansätze bei mehreren Spielen oft stark ähneln. So wurde in der Lösung des Damespiels eine Endspieldatenbank verwendet [Schaeffer 2007], wie sie in der Schachprogrammierung seit Jahren üblich ist [Steinwender 1995]. Weitere Konzepte wie Transpositionstabellen, Eröffnungsdatenbanken, lernbare Evaluationsfunktionen lassen sich für ein breites Sprektrum von Spielen einsetzen.

Erfolgreiche Computerspieler, die einen großen Teil ihrer Spielstärke aus einer für das Spiel konzipierten Hardware beziehen, wie beispielsweise Deep Blue [Campbell 2002], verzerren ein wenig die Vergleichbarkeit der Intelligenz dieser Spieler. Eventuell liegen sogar bessere Spieler vor, die allerdings nur auf herkömmlicher Hardware realisiert wurde. Dabei führt vor allem die Weiterentwicklung der KI-Komponente zu zusätzlichem Verständnis und Entwicklungen intelligentem Computerverhaltens. Einen Rahmen für solche Forschungen zu schaffen ist das Ziel des *jGameAI*-Frameworks.

Ein schneller Einstieg in die Programmierung künstlicher Intelligenz soll ermöglicht werden, ohne sich an der zeitintensiven Realisierung von Komponenten

aufzuhalten, die nicht direkt mit der Intelligenz des Spielers zusammenhängen. Die Funktion dieser Komponenten soll von einer breiten Anzahl an generischen Basisklassen übernommen werden.

Durch die Ausgliederung aller lernbaren Funktionen der Suche und Bewertung ist es dem Spieler möglich, diese einzelnen mit eigenen Lernmethoden zu trainieren und so die Spielstärke zu erhöhen. Um die Anpassungen der Koeffizienten beim Lernen besser überwachen und nachvollziehen zu können, werden diese auch graphisch veranschaulicht.

Ein weiteres Ziel des Frameworks ist es, die Komponenten der Spiele und der Interaktion zwischen Spielern und der Spielwelt möglichst abstrakt und generisch zu halten und somit für alle zukünftigen eingebundenen Spiele verwendbar zu sein.

8.1.1 Projekt-Entwicklung

Das Projekt *jGameAI* wurde als Vorarbeit zu dieser Diplomarbeit im Frühjahr 2008 gestartet. Grundlage für die Idee waren die erfolgreichen Sucherweiterungen, die im Rahmen des Schachprogramms *FUSc#* entwickelt wurden. Es wurde untersucht, in wiefern diese Erkenntnisse auch für eine breite Klasse von Spielen verwendbar sind.

Um den Einsatz auf möglichst vielen Betriebsystemen zu ermöglichen, wurde die Programmiersprache *java* gewählt. Die möglichen Geschwindigkeitsverluste fallen nicht ins Gewicht, da der Fokus der Lösungen auf die reine Leistung der künstlichen Intelligenz gelegt wird.

Es handelt sich bei *jGameAI* um ein großes openSource-Projekt, welches erfolgreich unter SourceForge angemeldet wurde und von der AG-Spieleprogrammierung der Freien Universität Berlin weitergeführt wird [Spiele AG (URL)]. Weitere Spiele und Lösungskonzepte werden kontinuierlich in wissenschaftlichen Arbeiten in das Framework eingebunden.

Mittlerweile umfasst das Projekt mehr als 150 Programmcodedateien und ermöglicht wissenschaftliche Forschungsarbeiten für mehr als fünfzehn Spiele. Unter Abschnitt 8.3 ist eine Liste aller bisherigen und laufenden Projekte zu finden.

8.1.2 Übersicht Softwarearchitektur

Das Framework wurde in mehrere logische Pakete eingeteilt. In der Tabelle 8.1 ist eine Übersicht aller Pakete inklusive einer kleiner Beschreibung aufgelistet.

Paket	Beschreibung
interfaces	Schnittstellen aller im Framework verwendeten Klassen
game	Basisklassen für alle eingebundenen Spiele
game.communications	Interne Kommunikation zwischen den Komponenten des Spiels
game.gui	Graphische Komponenten der allgemeinen GUI von Spielen
game.search	Implementierung des allgemeinen Suchalgorithmus
game.evaluation	Basisklassen für die Bewertungsfunktionen
game.learning	Implementierung des allgemeinen Lernalgorithmus
game.tournaments	Turniermodi zum Abhalten mehrerer Trainingspartien

Tabelle 8.1: Übersicht aller Pakete des Frameworks

8.2 Die Pakete

8.2.1 Die Basisklassen für alle Spiele und Spieler

An Spielen nehmen Spieler (*IPlayer*) teil, die in einer durch die Spielregeln bestimmten Welt (*IState*) nach vorgeschriebenen Mustern über Spielzüge (*IStateAction*) agieren. Spieler melden ihren Willen an einem Spiel teilzunehmen an und erhalten spielspezifische Parameter, wie etwa die zur Verfügung stehende Zeit, Spielerfarbe, Spielerreihenfolge, Startkapital usw. Sobald das Spiel begonnen hat, werden die Spieler ihren Positionen entsprechend der Reihe nach gebeten, Züge auszuführen. Für die Kommunikation zwischen Spiel und Spieler ist häufig ein Protokoll notwendig.

Nach einer Suche, die Wissen über die Spielwelt verarbeitet, treffen die Spieler ihre Entscheidungen und geben diese dem Spiel bekannt. Der Zug wird ausgeführt und alle Spieler werden über den neuen erreichten Spielzustand informiert. Die zentralen Programmkomponenten und ihre Kommunikation untereinander werden in Abbildung 8.1 dargestellt.

Die Basisklassen sollen dabei einen möglichst großen Teil der Funktionalität der Programmkomponenten übernehmen. Dies umfasst alle verallgemeinerbaren Aufgaben, die den Implementierungen aller Spieler und allen Spielen gemeinsam ist.

Alle Klassen des Projekts werden in ihren jeweiligen Schnittstellen beschrieben. Dabei wurden die Funktionen, die allgemein für alle Spiele gültig sind, bereits in den Basisklassen implementiert. Die restlichen werden als *abstract* deklariert und müssen während des Einbindens neuer Spiele beim Vererben gefüllt werden:

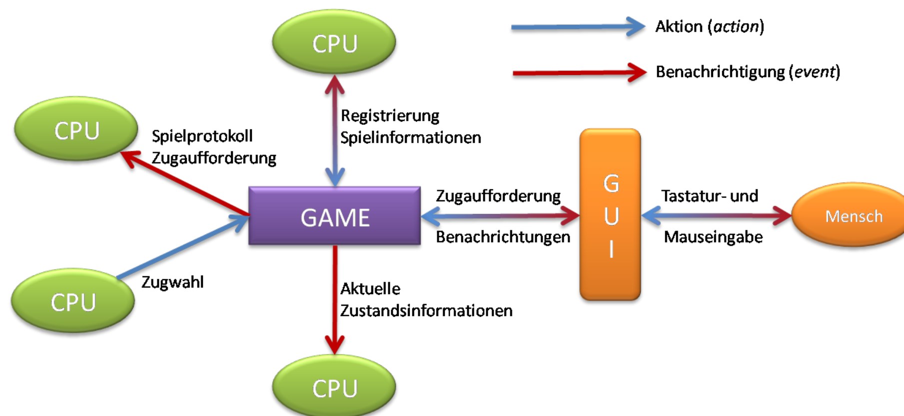


Abbildung 8.1: Die Abstraktion von Computerspielen in jGameAI

8.2.1.1 Die Grundklasse für alle Spiele: Game

Die Klasse verwaltet das Kommunikationssystem, wodurch alle Akteure und spielbeobachtenden Komponenten (*IGameObserver*) über die Vorgänge des Spiels informiert werden. Sie besitzt genau eine Instanz von der im Spiel verwendeten Zustandsklasse. Alle Spieler melden sich einmalig beim Spiel an und werden von da an im Einklang mit dem Spielprotokoll über das Spielgeschehen informiert.

8.2.1.2 Die Zustandsverwaltung eines Spiels

Als erstes gilt es, beim Einbinden neuer Spiele die Spielwelt selbst zu beschreiben. Dies wird durch das Ableiten der Klasse State übernommen. Eine Instanz dieser Klasse entspricht einem bestimmten Zustand, initial dem Startzustand eines Spiels. Auf diesem lassen sich Züge ausführen, worauf hin sich interne Parameter aktualisieren und die Instanz anschließend den Folgezustand darstellt.

In einem State werden auch alle beteiligten Spieler (*allPlayers*) geführt, sowie alle bisher ausgeführten Züge (*history*) gespeichert. So ist es möglich im Nachhinein eine Partie erneut durchzuspielen, was für das offline-maschinelle Lernen wichtig ist. Zusätzlich werden die prognostizierten Spielverläufe der Spieler - falls vorhanden - zu jedem Zeitpunkt t der Partie mitgeführt. Der Zeitpunkt t entspricht dabei der Anzahl der bisher ausgeführten Spielzüge und stattgefundenen Zufallsereignisse seit Spielbeginn.

Bei der Zuggenerierung gilt es zu berücksichtigen, aus welcher Sicht diese aufgerufen wird. In Spielen mit unvollständiger Information muss unterschieden

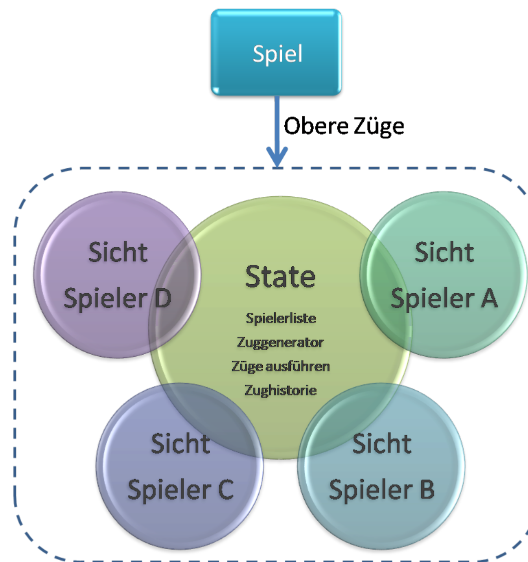


Abbildung 8.2: Die Funktionen der Klasse State und die unterschiedlichen Ansichten der Spielwelt. Obere Züge erfolgen durch die Spielinstanz; untere während der Spielerspezifischen Suche innerhalb der dem Suchenden zur Verfügung stehenden Informationen.

werden, welche Information welchen Spielern zur Verfügung stehen. Daher erhält die Suche für das Aufspannen des Suchbaumes die Zuglisten nur aus Sicht des suchenden Spielers. Es kann während der Suche dadurch zur Verwendung von verdeckten Zügen kommen, die wie Zufallsereignisse behandelt werden müssen.

Die Zustandsklasse erfährt nicht, ob es sich bei dem ziehenden Spieler um einen menschlichen oder Computerspieler handelt. Auch besitzt die Klasse State kein Wissen darüber, ob es sich bei einem Spielzug um einen suchinternen ("unterer Züge") oder real ausgespielten Spielerzug ("obere Züge") handelt.

Untere Züge werden vom Suchalgorithmus direkt auf dem State ausgeführt, während obere durch die Spielinstanz an den State weiter gereicht werden. Nur nach oberen Zügen wird überprüft, ob sich sich der State in einem legalen Zustand befindet. Damit können zwar illegale Zustände während einer Suche entstehen. Dieser Schritt ist aber notwendig, um auch verdeckte Züge in die Suche integrieren zu können. Spieler stellen hier eine statistische Vermutung über die Zugmöglichkeiten der Gegner an und führen dadurch auch zwangsweise die Züge mit, die real nicht möglich sind.

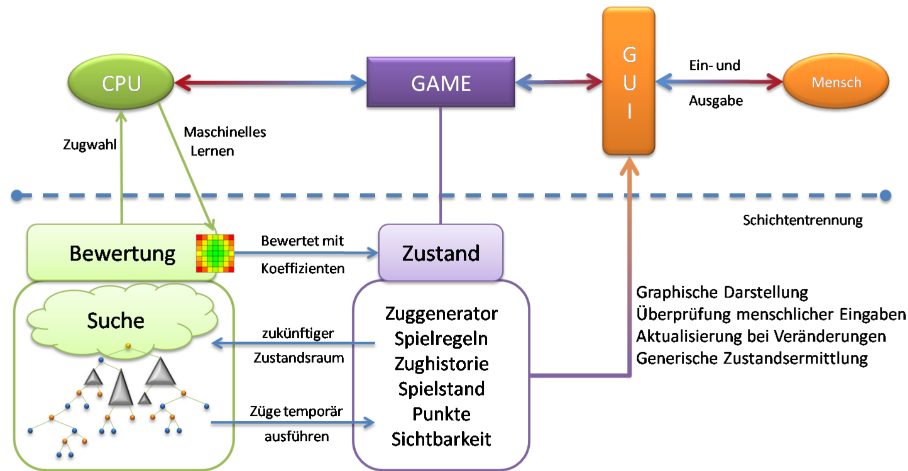


Abbildung 8.3: Modellierung von Computerspielern in Framework

8.2.1.3 Modellierung eines allgemeinen Spielers: Player

Hier befindet sich eine generische Beschreibung von Computer- und menschlichen Spielern (*IPlayer*). Sie melden sich über einen GameServer bei einem Spiel an und werden von da an über alle Veränderungen der Spielwelt informiert und in bestimmten Momenten des Spiels zur Eingabe eines neuen Zuges aufgefordert. Diese Klasse implementiert alle Funktionen des lernfähigen Spielers.

8.2.1.4 Generische Bewertungsfunktion für Zustände

Die Klasse Evaluation übernimmt die interne Bewertung von Spielzuständen für einen Spieler. Es wird dabei üblicherweise nach bestimmten Stellungskriterien gesucht und anhand der generisch abgespeicherten und verwalteten Gewichtsvektoren bewertet. Für jeden bekannten Gegner wird ein Parametersatz mitgeführt und später parallel zu den eigenen maschinell angepasst.

8.2.1.5 Graphische Darstellung von Spielen

Eine generische graphische Oberfläche (*IGUI*), die einen *JPanel* [Java (URL)] für Zeichnungen der Spielwelt zur Verfügung stellt. Alle Ereignisse des Spiels werden hier aufgefangen und können bei der Darstellung berücksichtigt werden. Sie führt zu jedem Zeitpunkt eine Liste mit den aktuell legalen Spielzügen mit, um somit menschliche Eingaben auf ihre Regelkonformität überprüfen zu können.

8.2.2 Das interne Kommunikationssystem

Für die Kommunikation zwischen den Akteuren eins Spiels und den spielbeobachtenden Komponenten wurde ein *action/event*-System konzipiert und implementiert. In einem solchen System werden nur auf einer zentralen Spielinstanz über eine einzige Funktion kodierte Aktionen ausgeführt. Dabei unterscheiden sich die Aktionen nur durch ihre einzigartigen *actionId*. Die Liste aller generischen Aktionen:

```
public final static int GAME_ACTION_START                          = 1000;
public final static int GAME_ACTION_PAUSE                          = 1001;
public final static int GAME_ACTION_RESUME                         = 1002;
public final static int GAME_ACTION_END                            = 1003;
public final static int GAME_ACTION_TOGGLE_DEBUG                   = 1004;
public final static int GAME_ACTION_EXIT                           = 1005;
public final static int GAME_ACTION_SEARCH_INTERRUPT               = 2001;
public final static int GAME_ACTION_SEARCH_START                   = 2002;
public final static int GAME_ACTION_LASTMOVE_UNDO                  = 3001;
public final static int GAME_ACTION_PERFORM_MOVE                   = 3002;   *
public final static int GAME_ACTION_COEFFICIENTS_SAVE              = 4001;
public final static int GAME_ACTION_COEFFICIENTS_TRAIN             = 4002;   *
public final static int GAME_ACTION_SEARCHDEPTH_CHANGE             = 4003;   *
public final static int GAME_ACTION_SET_GAMETIME                   = 5001;   *
public final static int GAME_ACTION_INFO_SHOW                      = 6001;   *
public final static int GAME_ACTION_INFO_RESET                     = 6002;
public final static int GAME_ACTION_PRINT                          = 7001;
public final static int GAME_ACTION_SIMULATIONMODE_TOGGLE          = 7002;
public final static int GAME_ACTION_ADD_PLAYER                     = 8001;   *
public final static int GAME_ACTION_SWITCH_PLAYERS                 = 8002;   *
public final static int GAME_ACTION_REMOVE_PLAYERS                 = 8003;
public final static int GAME_ACTION_TRIGGER_EVENT                  = 9001;   *
mit einem * gekennzeichnete Aktionen besitzen noch weitere Übergabeparameter
```

Mit Ereignissen informiert das Spiel alle Beobachter über das aktuelle Spielgeschehen und jede Veränderung an Spieleinstellungen und Zuständen. Die Ereignisse werden vom Spiel an alle Beobachter weitergeleitet. Eventuell werden die Beobachter dadurch veranlasst, eigene neue Aktionen durchzuführen. Ein typisches Beispiel für ein Ereignis ist das GAME_EVENT_STATEACTION_PERFORMED-Ereignis, welches nach jedem von einem Spieler ausgespielten Zug ausgelöst wird und zum Beispiel zu einem Neuzeichnen der graphische Oberfläche führt.

Es gilt aber: nur die Spielinstanz selbst kann neue Ereignisse auslösen, da nur hier zentral der aktuelle Spielzustand verwaltet werden soll. Jede Klasse,

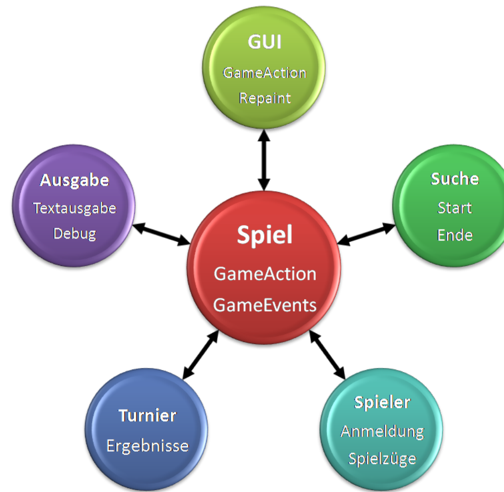


Abbildung 8.4: Die am Kommunikationssystem beteiligten Komponenten

welche die Schnittstelle *GameObserver* implementiert, kann einem Spiel als *GameListener* hinzugefügt werden und wird fortan über alle Ereignisse informiert:

```
public final static int GAME_EVENT_GAME_STARTED                    = 1001;
public final static int GAME_EVENT_GAME_ENDED                      = 1002;
public final static int GAME_EVENT_GAME_PAUSED                     = 1003;
public final static int GAME_EVENT_GAME_RESUMED                    = 1004;
public final static int GAME_EVENT_GAMETIME_CHANGED                = 1005;  *
public final static int GAME_EVENT_GAME_PRINTED                    = 1006;
public final static int GAME_EVENT_GAME_PREGAME_PHASE_STARTED      = 1007;
public final static int GAME_EVENT_GAME_AFTERGAME_PHASE_STARTED    = 1008;
public final static int GAME_EVENT_GAME_EXITED                     = 1009;
public final static int GAME_EVENT_GAME_DEBUGMODE_TOGGLED          = 1010;
public final static int GAME_EVENT_STATEACTION_PERFORMED           = 2001;
public final static int GAME_EVENT_SEARCH_INTERRUPTED              = 2002;
public final static int GAME_EVENT_SEARCH_FINISHED                 = 2003;
public final static int GAME_EVENT_SEARCH_STARTED                  = 2004;
public final static int GAME_EVENT_COEFFICIENTS_SAVED              = 3001;
public final static int GAME_EVENT_COEFFICIENTS_TRAINED            = 3002;
public final static int GAME_EVENT_SEARCH_DEPTH_CHANGED            = 3003;  *
public final static int GAME_EVENT_LAST_MOVE_UNDONE                = 4001;
public final static int GAME_EVENT_SIMULATION_MODE_TOGGLED         = 4002;
public final static int GAME_EVENT_PLAYER_ADDED                    = 4003;  *
public final static int GAME_EVENT_PLAYERS_SWITCHED                = 4004;
public final static int GAME_EVENT_PLAYERS_REMOVED                 = 4005;
public final static int GAME_EVENT_TOGGLE_PEVAL_DISPLAYED          = 4006;
public final static int GAME_EVENT_INFO_SHOWN                      = 5001;  *
public final static int GAME_EVENT_INFO_RESETTED                   = 5002;
public final static int GAME_EVENT_PLAYERSTIME_UPDATED             = 6001;  *
mit einem * gekennzeichnete Aktionen besitzen noch weitere Übergabeparameter
```

In einigen Spielen kann es zu Interaktion zwischen zwei oder mehreren Spielern in Form von Handel, Tausch von Ressourcen oder Ausspielen von Eigenschaftskarten kommen. Der Einfachheit halber sollte diese ebenfalls über die zentrale Instanz des Spiels verlaufen, um eine möglichst für alle Spiele einheitliche Kommunikation zu gewährleisten. Dafür eignet sich das Überschreiben von *GameAction* mit eigenen Parametern. Soll eine eigene Benachrichtigung erfolgen, kann dies mit einem überschriebenen *GameEvent* geschehen.

8.2.3 Generische graphische Oberflächen

Zu den Komponenten eines Spiels, die graphisch dargestellt werden können, zählen die Spielwelt, die Spieler mitsamt ihren Gewichtsvektoren, die Eingabemöglichkeiten für menschliche Spieler und eine Ausgabe aller vom Spiel ausgelösten Ereignisse. Bis auf die Darstellung der Spielwelt selbst können alle Komponenten generisch erzeugt und angezeigt werden. Dies wird in *jGameAI* mit Hilfe des *Java.swing*-Pakets [Java (URL)] durchgeführt.

Ein Spieler wird dabei als eine Liste von Registerkarten dargestellt, die jeweils eine für die eigenen und alle gegenerischen Evaluationsklassen beinhaltet. Zur besseren Unterscheidbarkeit können Spieler mit Icons initalisiert werden, die in allen graphischen Darstellungen neben dem Namen verwendet werden. Die Ansicht der Evaluationsklasse besteht selbst wiederum aus Registerkarten, die für jede Stellungsklasse eines Spielers die Gewichtsvektoren darstellt. Eine Gruppierung der Gewichtsvektoren ist nicht zwingend notwendig, hilft aber die Darstellung und interne Verwendung im Quellcode zu vereinfachen. Zudem bietet es sich bei vielen Spielen aufgrund regelmässiger Muster in der Spielwelt an.

Zusätzlich zu den Evaluationsklassen und der mit der graphischen Darstellung möglichen Manipulation und Überwachung der Daten, stellt der Spielerpanel noch weitere Debugfunktionen zur Verfügung. So kann der Spieler zu jedem Zeitpunkt aufgefordert werden, auch wenn er nicht an der Reihe ist, die aktuelle Stellung zu bewerten, nach einem besten Zug zu suchen oder eine Stellung mit den Gewichtsvektoren einer festgelegten und nicht vom Spieler selbst ermittelten, Stellungsklasse zu bewerten. Diese Funktionen werden als Buttons realisiert, welche sich für weitere speziellere Bedürfnisse erweitern lassen.

Für die Ausgabe aller Spielereignisse wurde eine einfache gepufferte Textausgabe gewählt, welche die neuesten Meldungen zuerst anzeigt. Zum Auslösen von Spielaktionen, wie etwa das Starten, Unterbrechen und Beenden eines

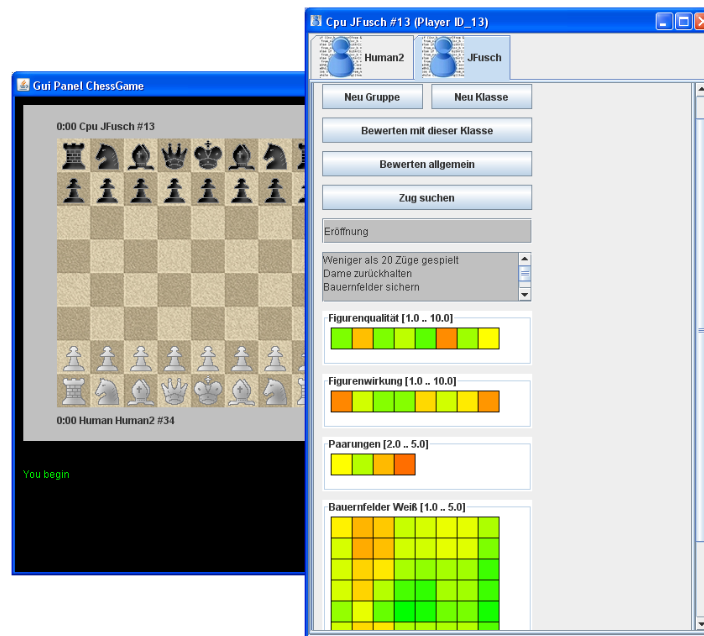


Abbildung 8.5: Der Spielerpanel des Schachprogramms JFusch. In der Stellungsklasse, welche die Eröffnung beschreibt, sind mehrere Koeffizientengruppen über die graphische Darstellung angelegt worden.

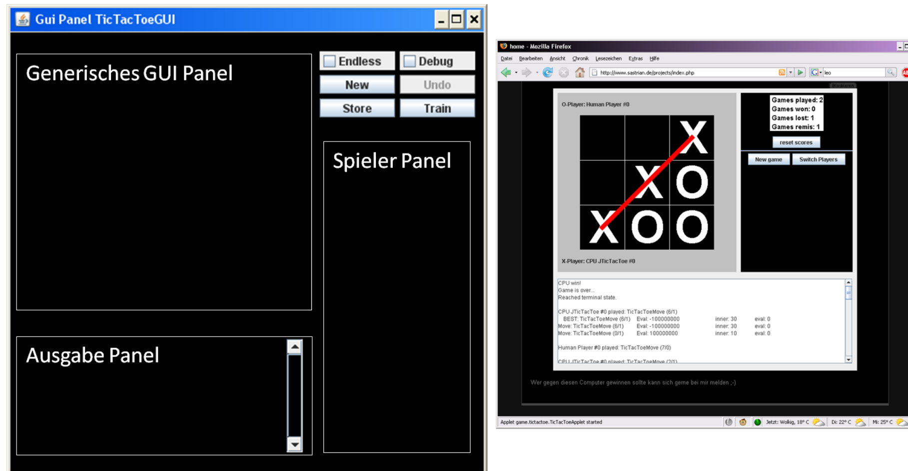


Abbildung 8.6: Die allgemeine graphische Darstellung von Spielen in jGameAI. Links zusammengeführt in einem Fenster, rechts als Webapplet. Alle Komponenten, bis auf die Spielweltdarstellung, hier im rechten Bild das TicTacToe-Gitter, lassen sich generisch erzeugen und sind in ihrer Funktionalität im Framework integriert. (Hier mit einer veralterten Version der Darstellung)

Spieles steht ein mit Buttons gefüllter Panel zur Verfügung. Gemeinsam mit dem Spielweltpanel, in welches die Spieleentwickler die Darstellung der Spielwelt zeichnen, werden alle Komponenten gruppiert und in einem Fenster gemeinsam dargestellt. Alternativ kann statt einem Fenster für Webanwendungen ein Applet verwenden. Diese Schritte lassen sich generisch umsetzen. Somit steht jedem Spiel die graphische Darstellung von Grund aus zur Verfügung.

8.2.4 Suchen nach besten Zügen

Die Suche wird immer von der zentralen Spielinstanz ausgelöst, indem der als nächste ziehende Spieler aufgefordert wird, seinen nächsten Zug auszuführen. Daraufhin initialisiert der Spieler die allgemeine Suche, startet sie und liefert eine positive Rückmeldung, sobald diese abgeschlossen ist. Der gefundene Zug wird anschließend durch die Spielinstanz beim Spieler abgeholt und auf dem internen Zustand ausgeführt. Der Suchvorgang und alle verwendeten Aktionen und Ereignismeldungen werden in Abbildung 8.7 veranschaulicht.

Da der erste Zug einer gefundenen Strategie dem nächsten Zug des suchenden Spielers entspricht, darf es sich nicht um ein Zufallsereignis handeln. Dazu muss die Spielinstanz wissen, ob sich der interne Spielzustand gerade an einem

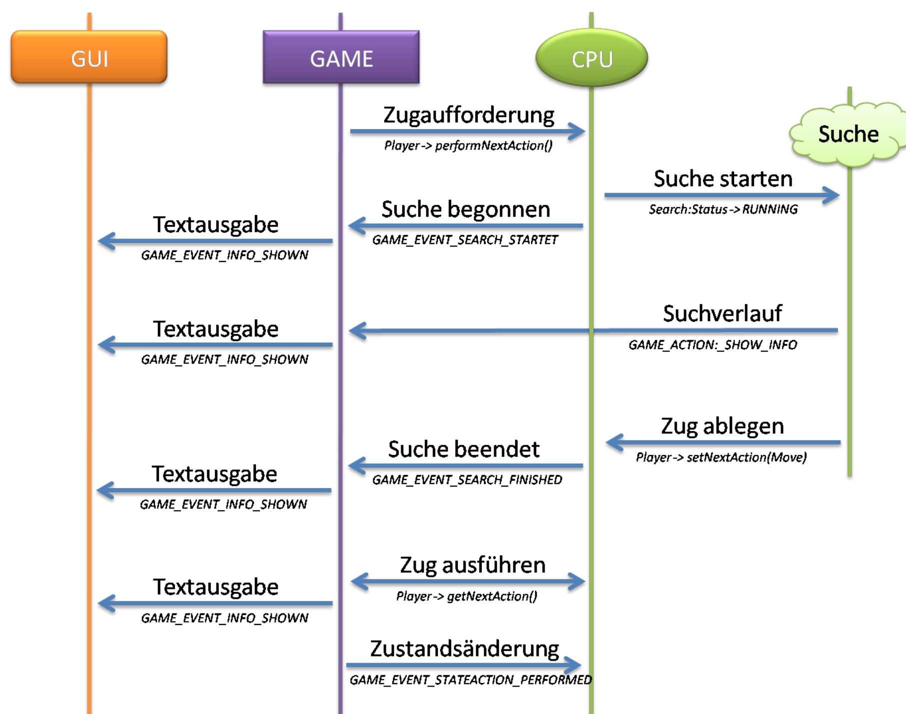


Abbildung 8.7: Das Kommunikationsprotokoll der Suche

Zufallsknoten befindet. Solange dies der Fall ist, wird der interne Zustand aufgefordert, das nächste Zufallsereignis entsprechend der Wahrscheinlichkeitsverteilung zu erzeugen und intern auszuführen. Erst anschließend wird der aktuelle Spieler mit der Suche nach einem neuen Zug beauftragt.

Auf der Suche nach ihrem nächsten Zug benötigen die Computerspieler eine Programmschnittstelle zu dem aktuellen Spielzustand. Über diese wird ihnen mitgeteilt, welche Züge und somit Folgezustände akzeptiert werden. Die während der Suche verwendeten Zuglisten müssen aus Sicht des suchenden Spielers generiert werden. Dafür realisiert die Suche die Schnittstelle des Spielerbeobachters (*IPlayerObserver*) und führt somit den suchenden Spieler als eigene Variable (*thePlayer*) mit.

Die Suche ist eingebettet in einen sie überwachenden Thread, die SearchEngine (*ISearchEngine*). Dieser Prozess überprüft alle 50 Millisekunden mögliche Zeitüberschreitungen und den aktuellen Zustand der Suche. Sollte diese beendet sein, erhält die Spielinstanz eine Meldung. Muss die Suche zum Beispiel wegen Zeitüberschreitung unterbrochen werden, so wird der interne Zustand der Suche von der SearchEngine auf *INTERRUPT* gesetzt. Im nächsten rekursiven Aufruf der Suche bricht diese dann ab. Der bisher beste gefundene Zug wird dem Spieler übergeben.

8.2.5 Bewerten von Zügen durch die Computer-Spieler

Von einer Stellungsbewertung (*IEvaluation*) wird erwartet, dass Sie für einen beliebigen Zustand und Spieler einen ganzzahligen Wert zurückliefert (*IEvaluation->evalStateForPlayer*), der aussagt, wie günstig die aktuelle Spielsituation für diesen aus Sicht des Bewertenden Spieler ist. Im Idealfall sollten dafür die allgemeinen Zugriffsfunktionen der Klasse State kombiniert mit den Konstanten der Konfigurationsklasse des aktuellen Spiels ausreichen, um alle benötigten Informationen zu ermitteln. Werden besondere Funktionalitäten benötigt, kann durch Casting des State Objekts hin zur spezifischen Spieldarstellung der erweiterte Satz an Zugriffsfunktionen verwendet werden.

Jedem Spieler steht ein Array zur Verfügung, die für jeden am aktuellen Spiel teilnehmenden Spieler ein Evaluationsobjekt bereithält. Wie die Bewertung funktioniert, muss nach aussen hin nicht bekannt gegeben werden. Für jeden neuen Gegner werden Kopien der eigenen Bewertung inklusive aktueller Gewichtsvektoren erzeugt und abgespeichert. Bei zukünftigen Aufeinandertreffen mit diesen Gegnern werden die zuletzt gespeicherten Vektoren erneut

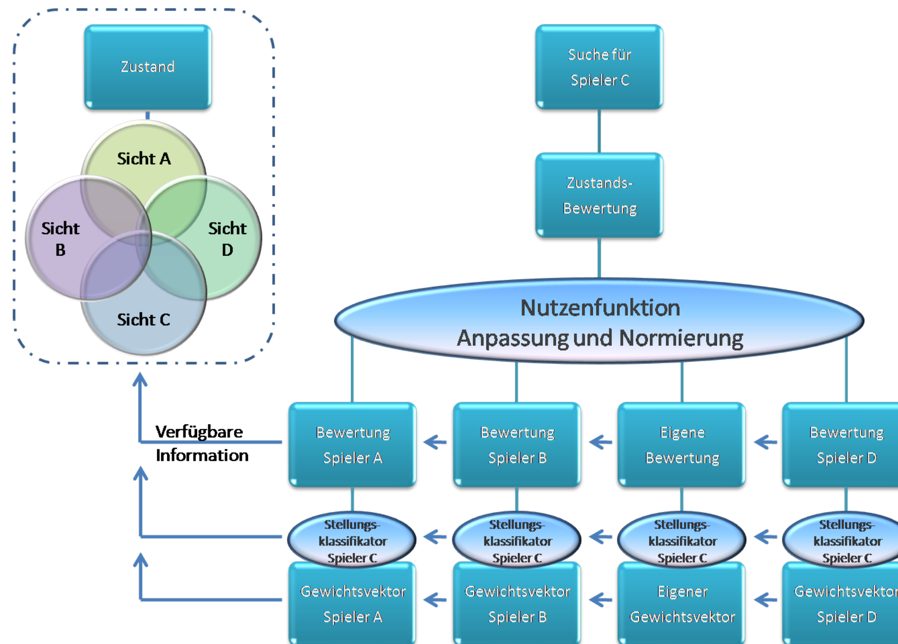


Abbildung 8.8: Evaluations-Beispiel aus Sicht eines Spielers C

verwendet. Dadurch ist es möglich, langfristige Gegnerprofile zu erstellen und mitzuführen.

Die verwendeten Gewichtsvektoren können zur logischen Gruppierung zusammengelegt werden. Dazu werden sie als Werte in einer EvalCoefficientsGroup zusammen abgespeichert. Jede Gruppe besitzt einen Wertebereich (range_max, range_min), Name und Beschreibung und eigene Lernparameter (Lernrate und Schrittgröße). Die Einteilung der Gewichtsvektoren ist in vielen Spielen sinnvoll und erleichert die Verwendung innerhalb des Quellcodes, wo sie als Indizes einer Werteliste durchlaufen werden können.

Die nächste logische Einteilung erfolgt durch Zuordnung der Gewichtsvektoren zu unterschiedlichen Stellungsklassen EvalClassCoefficients. Für jede Stellungsklasse kann je nach Bedarf ein unterschiedlicher Satz an Koeffizientengruppen vorliegen. Auch die Stellungsklassen besitzen zusätzliche Informationen wie den Wertebereich, Name und Beschreibung und übergeordnete Lernraten (so kann das Lernen für die Eröffnungsstellungen verhindert werden, wenn Eröffnungsbücher verwendet werden).

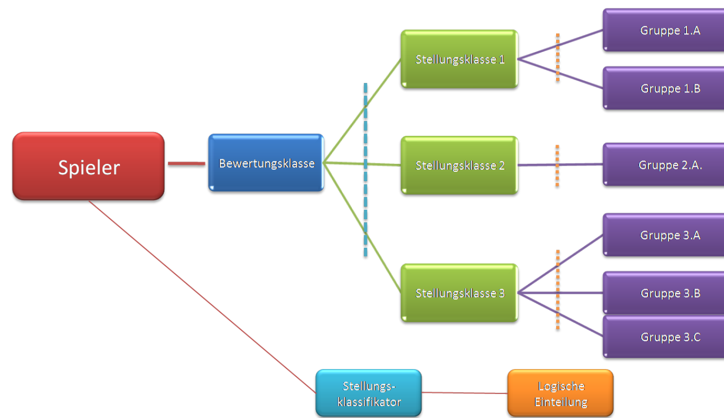


Abbildung 8.9: Die hierarschiche Einteilung der Evaluationskomponenten eines Spielers

Das Speichern, Zuordnen und Laden aller Evaluationskomponenten erfolgt generisch und wird von *jGameAI* übernommen. Dafür werden alle Komponenten in einem eigenen einfachen Dateiformat lokal auf der Festplatte abgelegt und beim Laden eines Spielers erstellt. Der interne Zugriff innerhalb des Quellcodes auf die Daten erfolgt über die in den Schnittstellen beschriebenen Funktionen.

Durch die Trennung der Gewichtsvektoren vom Quellcode ist es möglich einen mit *jGameAI* trainierten Spieler für weitere Spielerealisierungen bereit zu stellen. Diese können sogar in anderen Programmiersprachen entwickelt worden sein und eigene Suchalgorithmen verwenden. Somit ist der kleine Nachteil der Geschwindigkeitseinbußen durch die Verwendung der Programmiersprache *Java* und ein allgemein gehaltenes Konzept leicht zu ertragen, da das Erlernte auch für speziellere Anwendungen verwertbar bleibt.

8.2.6 Turniere und Trainingspartien

Um unterschiedliche Computergegner gegeneinander antreten zu lassen, wurde ein generisches Turniersystem entwickelt. Ein Turnier (*GameCluster*) besitzt eine Menge von angemeldeten Spielern, die nach einem vorgegebenen Begegnungsmuster gegeneinander in einzelnen Partien (*Match*) antreten. Ist eine Begegnung beendet, so informiert der Schiedsrichter (*MatchObserver*) das Turnier und übermittelt Ergebnisse, Punkte und wenn implementiert, weitere Informationen (*PlayerStats*).

Über einen Matchpanel lassen sich alle ausgegeben Informationen des Spiels verfolgen, die in der Mitte eines virtuellen Spieltisches angezeigt werden. Spieler

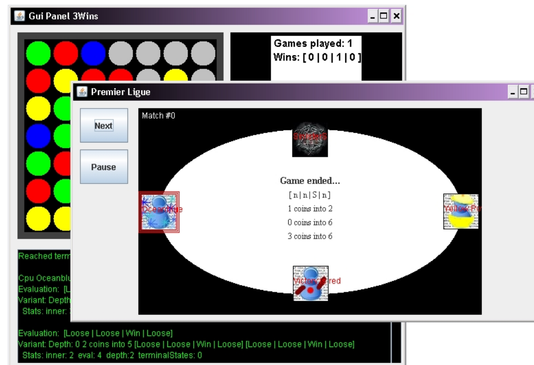


Abbildung 8.10: Der virtuelle Spieltisch einer Begegnung eines 3er! Turniers, das auch noch über eine gewöhnliche Oberfläche beobachtet wird. Diese kann dabei auf einem anderen Computer ausgeführt werden.

#	Spieler	Name	Spiele	Punkte	Differenz	S	U	N
1	Feynman	MasterBrain!	4	6	16	2	0	2
2		ProductivityStr...	4	3	14	1	0	3
3		AstroidsAllStar...	4	3	14	1	0	3
4		MoepPlayer	4	0	8	0	0	4

Abbildung 8.11: Eine generisch erstellte Gruppentabelle nach Durchspielen von 4 Begegnungen im Spiel Siedler von Catan. Rosa untermalt sind die Punkte des Grupenführenden.

werden dabei mit Hilfe eigener Icons gezeichnet. Parallel dazu kann ebenfalls jede GUI, die sich bei der Spielinstanz des Turnieres anmeldet, das Spiel beobachten.

Für manche Turniere müssen Tabellen mitgeführt werden, um zu ermitteln, welche Spiele ausscheiden und weiterkommen. Diese Tabelle wird generisch von der Oberklasse GameCluster erstellt und nach jedem Match aktualisiert. Einige Standardturnierformen wurden entwickelt: Ligamodus mit Hin- und Rückrunde, Ligamodus mit vorgegebener fester Anzahl an Spielen je Spieler (gut geeignet für Trainingsphasen), 16-er K.O. Turnier ohne Vorrunde und 32-er K.O Turnier mit acht Vierergruppen.

8.3 Aktuelle Projekte

Das jGameAI Framework befindet sich derzeit in einer lauffähigen und stabilen Version, die unter der Versionsnummer 1.0 bei SourceForge released wurde. Alle hier vorgestellten Projekte sowie weitere Information zu der Arbeitsgruppe Spieleprogrammierung der Freien Universität lassen sich unter den im Anhang aufgeführten Internetadressen auffinden [Spiele AG (URL)].

Erste wissenschaftliche Arbeiten konnten bereits im Rahmen des Framework veröffentlicht werden [Wang/M 2008, Czerwionka 2008]. Aus den aktuellen Projekten werden vorraussichtlich weitere folgen.

8.3.1 Populäre Brettspiele: Schach, Dame und GO

Als erstes Projekt wurde Schach in das Framework eingebunden - mit dem Hintergrund, das erfolgreiche Schachprogramm *FUSc#* zu integrieren. Idealerweise sollten dafür auch die bereits trainierten Koeffizienten eingebunden werden können. Dafür wurden generische Datei- und Programmformate für die Bewertungskoeffizienten entwickelt. Ziel des Schachprojekts ist es, bis zur Langen Nacht der Wissenschaften 2009 in Berlin das ursprüngliche Schachprogramm *DarkFUSc#* vollständig in das Framework als eigenen Spieler eingebunden zu haben.

Dame ist trotz der spieltheoretischen Lösung [Schaeffer 2007] weiterhin interessant für die wissenschaftliche Erforschung von Computerspielern. Zum einen, da mit dem Programm chinook nun ein unbesiegbarer Gegner vorliegt (bei der Startstellung beginnend), der sich gut für Trainingspartiern eignet. Des weiteren bietet sich Dame aufgrund einer relativ überschaubaren Spielkomplexität für experimentelle Ansätze an. Darunter fällt beispielsweise das selbständige Finden von spielrelevanten Stellungskriterien, wie in Abschnitt 7.2.2 beschrieben.

Als weiteres populäres rundenbasiertes 2S-Spiel mit vollständiger Information wurde das Spiel GO ins Framework eingebunden. Aufgrund der enormen Spielkomplexität ist man allerdings hier angewiesen, alternative Lösungsverfahren zu untersuchen. Der erfolgreiche Einsatz von Monte-Carlo-Methoden in der Lösung von GO [Wang/Y 2007] zeigt, dass dadurch starke Computerspieler entstehen konnten.

8.3.2 Die nächste Herausforderung: Poker

Das Spiel Poker stellt mit seiner Komplexität viele Herausforderungen an künstliche Spieler, die Pokerbots [Billings 2002]. Mit üblicherweise bis zu zwölf Ge-

Abbildung 8.12: Die Spiele Go, Schach und Dame im Framework *jGameAI*

genspielern und den Elementen Bluffen und unbekannter Spielausgang werden Computer mit bisherigen Lösungsmethoden überfordert. Statisch spielende Pokerbots werden schnell durchschaut. Um erfolgreich zu bluffen, müssen allerdings sehr sichere Siegwahrscheinlichkeiten berechnet und die Gegner gut analysiert werden, um im Mittel Gewinne zu erzielen [Schauenberg 2006].

Dies ist zum derzeitigen Stand für Mehrspieler-Situationen kaum möglich. Es verwundert daher nicht, dass die derzeitig stärksten Pokerbots, etwa das an der *University of Alberta* in Kanada entwickelte Programm *Polaris*, nur in Eins zu Eins Situationen kleinere Gewinne gegen Pokermeister erzielen können [Polaris (URL)]. In der Doktorarbeit des Projektleiters von *Polaris* werden unterschiedliche Lösungswege für das Spiel Poker vorgestellt und besprochen [Billings 2006]. Ziel des Pokerprojekts in *jGameAI* ist es, mehrere dieser Ansätze zu realisieren und mit eigenen Ideen zu erweitern.

Um den direkten Vergleich mit derzeitigen Pokerbots zu ermöglichen ist die Anbindung an die Schnittstelle der Software *Poker Academy* implementiert worden [Poker Academy (URL)]. Das ebenfalls in Alberta entwickelte Programm bietet ein Kommunikationsprotokoll an, mit dessen Hilfe unterschiedliche Spieler und Bots gegeneinander über das Internet antreten können.

Abbildung 8.13: Das Spiel Poker in *jGameAI* während der Preflop-Phase

8.3.3 Das Strategiespiel *Siedler von Catan* in jGameAI

Das Strategiespiel *Siedler von Catan* [Catan (URL)] ist durch die vielen Zufallskomponenten, wie etwa das Würfeln und Kartenziehen, allein mit Suchbaumbasierten Verfahren schwer zu lösen. In [Thomas 2002] werden allgemeine Lösungskonzepte vorgestellt, mit denen sich Spiele wie *Siedler von Catan* angehen lassen können.

Für eine Multi-Touch-Realisierung des Strategiespiels verwendete Miao Wang im Rahmen seiner Diplomarbeit das Spieleframework *jGameAI* [Wang/M 2008-2]. Das Spiel wurde bis auf die graphische 3D-Darstellung und die Multi-Touch-Funktionalität vollständig in *jGameAI* realisiert. Dadurch war eine Schnittstelle zum Einbinden eigener künstlicher Spieler geschaffen.

Neben einer reinen Brettbewertung müssen künstliche Spieler des Spiels auch weitere Funktionen anbieten, die zu einer Reduzierung des Suchraumes führen. So werden etwa anstatt aller Bau- und Handelmöglichkeiten nur die jeweils N besten im Suchbaum berücksichtigt. Dabei ist der Wert von N je nach Rechnerleistung individuell anpassbar.

Während der Vorlesung "Künstliche Intelligenz" wurde im Sommer-Semester 2008 an der Freien Universität ein Turnier ausgetragen, bei dem die künstlichen Spieler der Studenten gegeneinander antraten. Mit dem einheitlichen Wert von $N = 5$ wurden eine Spielersuchtiefe von 4 ermöglicht, die eine Spitzen-

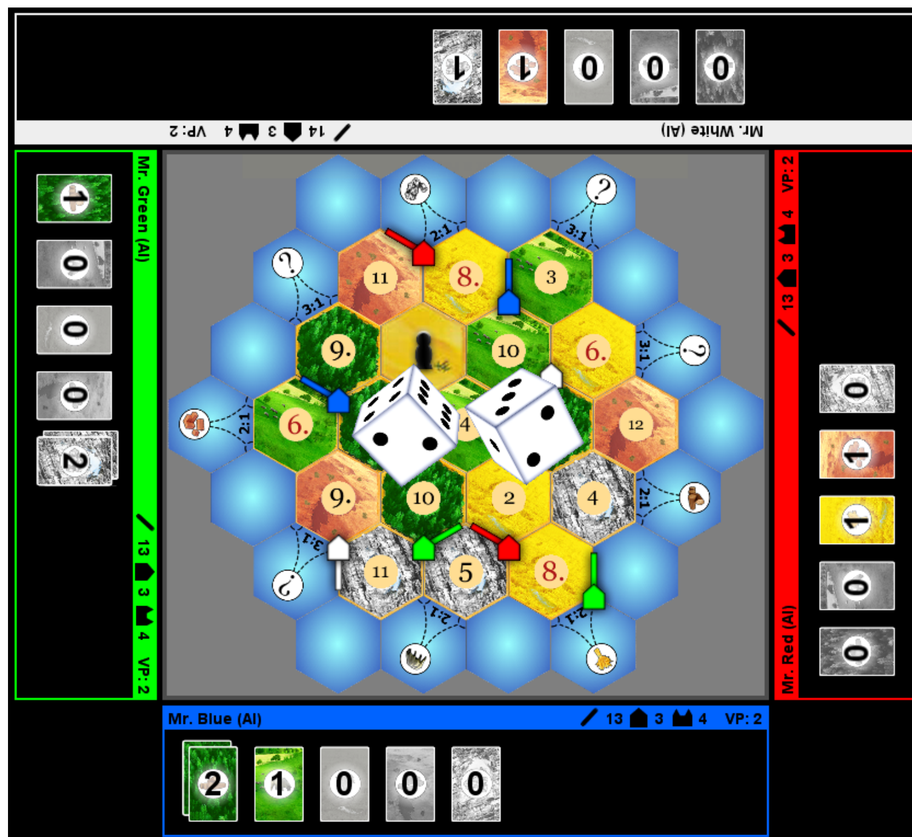


Abbildung 8.14: Die mit *jGameAI* entwickelte Multi-Touch-Anwendung *javaSettlers* von Miao Wang ([Wang/M 2008])

suchtiefe von bis zu 16 zukünftigen Handlungen erreichte. Weitere Informationen zu dem abgeschlossenen Projekt “javaSettlers” von Miao Wang werden in [Wang/M 2008-2, Wang/M 2008] aufgeführt.

Der Einsatz von maschinellen Lernmethoden war zu diesem Zeitpunkt noch nicht möglich, so dass sich ein erneutes Austragen des Turniers mit trainierten Spielern in naher Zukunft anbietet.

8.3.4 Echtzeitanwendungen: *jBirds* und *jAsteroidsAI*

Bisher wurden vor allem Rundenbasierte Spiele in das Framework eingebunden. Dies liegt sicherlich daran, dass die meisten Gesellschaftsspiele einem festgelegten Spielprotokoll unterliegen, das sich einfach mit Spielrunden realisieren lässt. Mit den Anwendungen *JBirds*, *JAsteroids* und *JLife* ist das Framework auch für Echtzeitanwendungen erweitert worden.

Echtzeitspiele lassen sich gut mit Agenten realisieren, die in der Spielewelt Handlungen unternehmen [Ertel 2008]. Dafür wurde in *jGameAI* die Erweiterung vorgenommen, dass Spieler ihre nächsten Züge mit Hilfe von Agenten suchen können statt mit dem verallgemeinerten Suchalgorithmus.

Die erste daraus resultierende Anwendung war eine Vorläuferversion des Programmes *jAsteroidsAI*, das einen künstlichen Spieler für das Spiel *Asteroids* modellierte [Czerwionka 2008]. Diese wurde erfolgreich bei einem Wettbewerb der Computerzeitschrift c’t eingereicht und konnte eine gute Plazierung erreichen [c’t contest (URL)].

Weitere Echtzeitanwendungen wurden eingebunden, die für spätere Forschungen und Neuentwicklungen evolutionärer Algorithmen [Weicker 2007] dienen sollen. Darunter die Anwendung *jBirds*, die das Schwarmverhalten von Vögeln simuliert, ohne die Bewegungsentscheidungen der Vögel zentral zu bestimmen. Jeder Vogel übernimmt dabei die Rolle eines Agenten, der sich in der Spielweltumgebung bewegt. Spätere Weiterentwicklungen sollen Vögel mit unterschiedlichen Parametern für die Entscheidungsfindung in einer Welt mit beschränkten Ressourcen konkurrieren lassen, so dass sich die Entwicklung evolutionärer Algorithmen direkt nachvollziehen lassen kann.

8.3.5 Die Spielwiese: 4-gewinnt!, 3-gewinnt! und 3-gewinnt!*

Zum schnellen Testen neuer Sucherweiterungen und zum Vergleich der daraus resultierenden unterschiedlichen Spieler eignen sich im Gegensatz zu Schach und GO einfachere Spiele wie 4-gewinnt! besser. Basierend auf dem Spielgitter

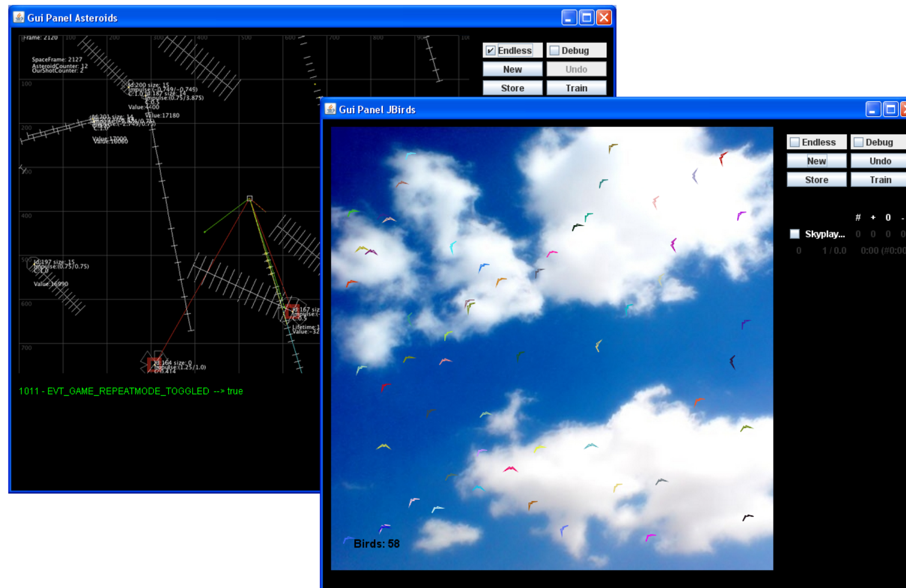


Abbildung 8.15: Die Echtzeitanwendungen *jAsteroidsAI* und *JBirds* in *jGameAI*

des bekannten Spiels 4-gewinnt! wurden zwei MS-Spiele erfunden, die sich zum Testen der Algorithmen auch für MS-Spiele mit und ohne Zufallsereignisse(n) eignen. Eine Erläuterung der Spielregeln dieser neuen Spiele befindet sich im Abschnitt 9.3.

Mit einer durschnittlichen Suchbreite von 7 - das Spielgitter besitzt 7 Spalten - handelt es sich um überschaubare Spiele, was die Suchraumkomplexität betrifft. Zusätzlich treten bei diesen Spielen viele transponierte Zustände auf, so dass früh Zusammenführungen identischer Zustände möglich sind. Ein guter Suchalgorithmus sollte daher ohne weiteres in Lage sein, Suchtiefen von $8 - 10$ mit aktuellen Heimcomputern zu erreichen.

Die Bewertungsfunktion der implementierten Spieler in *jGameAI* untersucht das Gitter nach vorhandenen offenen Zweier-Kombinationen (die noch zu einer Dreier-Kombination erweitert werden können) und nach Möglichkeiten solche zu erreichen, da sie eine für den Sieg notwendige Vorbedingung darstellen. Zusätzlich fließt jedes mit einer eigenen Figur besetztes Feld einem von den x/y Koordinaten abhängigen Wert in die Bewertung ein.

Tatsächlich reichen diese relativ trivialen Stellungskriterien aus, um bereits einen schwer zu besiegenden Spieler zu modellieren. Geplant ist mit diesen Spielen Verfahren zur selbständigen Entdeckung spielrelevanter Stellungskriterien,

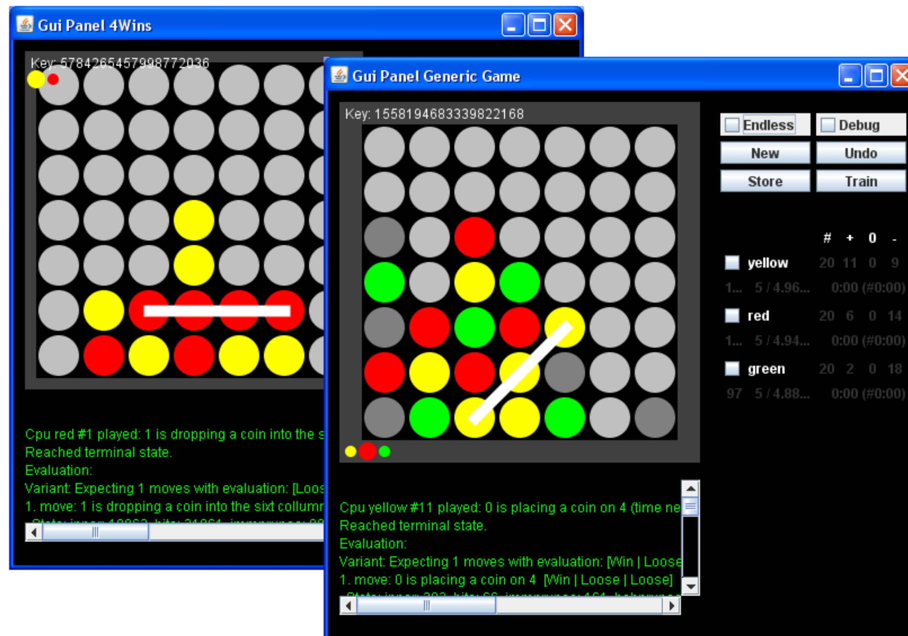


Abbildung 8.16: Die Testspiele 4-gwint und 3-gewinnt!* in *jGameAI*

wie sie in Abschnitt 7.2.2 beschrieben werden, zu untersuchen.

8.4 Ausblick und zukünftige Arbeiten

Im Rahmen der Arbeitsgruppe Spieleprogrammierung der Freien Universität Berlin wird das Framework ständig weiterentwickelt und mit neuen Spielen erweitert. Zur Versionsnummer $v1.0$ [jGameAI (URL)], sind bereits mehr als fünfzehn Spiele verfügbar. Es ist geplant, diese Menge stetig um weitere Spiele zu erweitern. Dies bietet sich an, um neue Konzepte für das maschinelle Lernen von Spielern zu testen oder zusätzliche allgemeine Suchbeschleunigungen einzubauen.

Des weiteren wird daran gearbeitet, die Spiele des Frameworks online bespielbar zu machen. Dafür wird ein Server-Client-Protokoll entwickelt, mit welchem die Kommunikation zwischen Remote und lokalen Spielern ermöglicht werden soll. Um den Server zu entlasten, soll es möglich sein sowohl Spiele als auch die Suche der gegnerischen KI lokal auszuführen. Dadurch bleibt das zentrale Anliegen, faire hardwareunabhängige Vergleichsmöglichkeiten von KI zu erhalten.

Darauf aufbauend bieten sich regelmässige Turniere an, die online ausgeführt werden können.

Um langfristige Spielerentwicklungen mitverfolgen zu können, ist die Realisierung von Spielerprofilen geplant. Damit lassen sich die gelernten Parameter jederzeit weltweit abrufen und untersuchen. Dies könnte interessant bei der Entwicklung eigener Computerspieler sein, da hier auf vorherige Ergebnisse zurückgegriffen werden kann.

Eine interessante Weiterentwicklung Suchbaumbasierter Verfahren ist das Einbinden sogenannter Monte-Carlo-Methoden. Diese bieten in Spielen die Möglichkeit, eine möglichst große Menge zufälliger Spielverläufe zu analysieren und somit eine Einschätzung der aktuellen Spielsituation vorzunehmen. Der Ansatz eignet sich gut für Spiele, die aufgrund ihrer Komplexität schwer für Suchbäume handzuhaben sind, wie etwa GO und Poker. In einigen aktuellen Projekten kommen diese Methoden bereits zum Einsatz, so dass es naheliegend ist, diese Erweiterung verallgemeinert zur Verfügung zu stellen und in den Suchalgorithmus einzubinden.

Kapitel 9

APPENDIX

9.1 Der $UPP - max^n$-Suchalgorithmus

```
// die rekursiv aufgerufene Suche
public float[] innerSearch() {

 // den eindeutigen Schlüssel der aktuellen Stellung ermitteln
 long currentKey = theState.getUniqueStateKey();

 // falls der Schlüssel ungleich 0 ist, überprüfen ob dafür bereits
 // eine Stellung vorliegt
 StateActionVariantNode oldNode = null;
 if (currentKey!= 0 && transpositionTable.containsKey(currentKey)) {
  oldNode = transpositionTable.get(currentKey);
 }

 // Überprüfen, ob wir einen transponierten Zustand wiedergefunden haben
 // wenn eine Stellung in der Tabelle vorliegt und das Wissen dieser
 // verwendet werden kann
 if (oldNode != null
   && oldNode.remainingPlayerDepth >= theCurrentNode.remainingPlayerDepth
   && oldNode.distanceToRoot <= theCurrentNode.distanceToRoot) {

  // interner Statistikzähler für Treffer in der Tabelle erhöhen
  ttHits++;

  // die Evaluationswerte der bereits besuchten Tabelle übernehmen
  theCurrentNode.evaluation = new float[nPlayers];
  for (int i = 0; i < nPlayers; i++) {
   theCurrentNode.evaluation[i] = oldNode.evaluation[i];
  }

  // Anderenfalls:
  // handelt es sich um eine Terminale Stellung?
 } else if (theState.isTerminalState()) {
```

```
 // interner Statistikzähler für terminale Stellungen erhöhen
 terminalCount++;

 // Spielergebnis dieser Stellung ermitteln
 theCurrentNode.evaluation = theState.getCurrentStanding();

 // liegt ein Unterbrechungssignal vor
 // oder sind wir an einem Blattknoten gelangt
} else if (theCurrentNode.remainingPlayerDepth <= 0
   || theStatus == ThreadBase.INTERRUPT) {

 // interner Statistikzähler für bewertete Knoten erhöhen
 evalCount++;

 // aktuelle Stellungn durch suchenden Spieler bewerten
 heCurrentNode.evaluation = thePlayer.evalGame(theState);

// Ansonsten liegt ein innerer Knoten vor
} else {

 // interner Statistikzähler für innere Knoten erhöhen
 nodeCount++;

 // wenn es sich um einen transponierten Zustand handelt
 if(oldNode != null) {
  // können wir die Zugliste übernehmen
  theCurrentNode.fillFromOldNode(oldNode);
 } else {
  // ansonsten Zugliste generieren
  theCurrentNode.fillNode();
 }

 // Überprüfen, ob wir einen deterministischen oder
 // Zufallsknoten erreicht haben

// wenn es sich um einen Zufallsknoten handelt
 if (theCurrentNode.departsFromProbabilityState()) {

  // Können wir Zufallsknoten-pruning anwenden?
  if(theCurrentNode.parentNode != null && theCurrentNode.index >0) {

   // Spieler des Elternknoten ermitteln
   int parentPlayer = theCurrentNode.parentNode.currentPlayer;

   // bisherige beste Bewertung für Elternspieler
   float parentPlayerMinEvalSoFar = theCurrentNode.parentNode.evaluation[parentPlayer];

   // die Wahrscheinlichkeit aller noch verbleibender Zufallsereignise
   float summedRestProbability = 1.f;

   // alle Zufallsereignisse durchspielen und Mindesbewertung untersuchen
   for (int i = 0; i < theCurrentNode.nActions; i++) {

    // i.ter Zufallsereignis ausführen
```

```
    theState.performAction(theCurrentNode.theActions[i]);

    // Restwahrscheinlichkeit korrigieren
    summedRestProbability -= theCurrentNode.theActions[i].getProbability();

    // Knotenpointer setzen
    theCurrentNode = theCurrentNode.theChilds[i];

    // rekursive Suche an Kindknoten ausführen
    float[] w = innerSearch();

    // Zufallsereignis zurücknehmen und Knotenpointer setzen
    theState.undoLastAction();
    theCurrentNode = theCurrentNode.parentNode;

    // die gefunde i.te Entscheidungsvariante inkls. Bewertung abspeichern
    theCurrentNode.add(w, i);

    // ist das beste noch erreichbare Ergebnise bereits
    // unter dem bisher besten gefundenen für den Elternspieler?
    if((theCurrentNode.evaluation[parentPlayer] + summedRestProbability*1)
        < parentPlayerMinEvalSoFar) {

    // Abschneiden möglich
    rndprunings++;

    // und die Suche abbrechen, da dieser Knoten nicht
    // in der optimalen Strategie liegt
    break;
   }
  }

 // Kein Zufallpruning möglich: berechnen des statistischen
 // Mittels über alle Zufallsereignisse
 } else {

  // alle Zufallsereignisse durchspielen
  for (int i = 0; i < theCurrentNode.nActions; i++) {

   // i.ter Zufallsereignis ausführen und Knotenpointer setzen
   theState.performAction(theCurrentNode.theActions[i]);
   theCurrentNode = theCurrentNode.theChilds[i];

   // der rekursive Aufruf der Suche
   float[] w = innerSearch();

   // i.ten Zufallsereignis zurücknehmen und Knotenpointer setzen
   theState.undoLastAction();
   theCurrentNode = theCurrentNode.parentNode;

   // die gefundene i.te Entscheidungsvariante inkl. Bewertung abspeichern
   theCurrentNode.add(w, i);
  }
 }
} else {
```

```
// Deterministischer Knoten:
// Alle möglichen Züge untersuchen
for (int i = 0; i < theCurrentNode.nActions; i++) {

 // den i.ten Zug ausführen und Knotenpointer setzen
 theState.performAction(theCurrentNode.theActions[i]);
 theCurrentNode = theCurrentNode.theChilds[i];

 // einen eventuellen Spielerwechsel feststellen
 theCurrentNode.currentPlayer = theState.getCurrentPlayerPosition();
 if (theState.hasCurrentPlayerChanged()) {
  theCurrentNode.remainingPlayerDepth--;
 }

 // der rekursive Aufruf der Suche
 float[] w = innerSearch();

 // den i.ten Zug zurücknehmen und Knotenpointer setzen
 theState.undoLastAction();
 theCurrentNode = theCurrentNode.parentNode;

 // wurde eine Verbesserung gegenüber der bisherigen optimalen
 // Strategie gefunden?
 if (w[theCurrentNode.currentPlayer] >
       theCurrentNode.evaluation[theCurrentNode.currentPlayer]) {

  // die Bewertung und Strategie des i.ten Kindknotens übernehmen
  theCurrentNode.updateEvaluation(i);

  // eine Debugausgabe zur Information
  if (theCurrentNode.remainingPlayerDepth == maxPlayerDepth) {
   theGame.performGameAction(new GameInfoAction(getStats()));
  }

  // immediate pruning untersuchen (Terminale Stellung mit eigenem Sieg gefunden)
  if(w[theCurrentNode.currentPlayer] >= 1) {

   // Abschneiden aller restlicher Bruderknoten
   for (int j = i + 1; j < theCurrentNode.nActions; j++) {
    theCurrentNode.remove(j);
   }

   // Statistikzähler für mit immediate pruning abgeschnittene Knoten erhöhen
   immprunings++;

   // Suche beenden
   break;
  }

  // paranoid pruning untersuchen (Verschlechterung des vorherigen Spielers)
  if(theCurrentNode.index > 0) {

   // den vorherigen Spieler ermitteln
  int parentPlayer = theCurrentNode.parentNode.currentPlayer;
```

```
     // paranoide Spieler: eine weitere Betrachtung des Knotens abbrechen?
     if(parentPlayer!=theCurrentNode.currentPlayer
      && theCurrentNode.parentNode.evaluation[parentPlayer] > w[parentPlayer]) {

     // Abschneiden aller restlicher Bruderknoten
     for (int j = i + 1; j < theCurrentNode.nActions; j++) {
      theCurrentNode.remove(j);
     }

     // Statistikzähler für mit paranoid pruning abgeschnittene Knoten erhöhen
     babprunings++;

     // Suche beenden
     break;
    }
   }

  } else {

    // der i.te Kindknoten wurde erfolglos abgearbeitet und kann entfernt werden
    theCurrentNode.remove(i);
   }
  }
 }

 // bearbeiteten Knoten in der Transpositionstabelle abspeichern
 if(currentKey != 0)
  transpositionTable.put(currentKey, theCurrentNode);

 // Bewertung des bearbeiteten Knoten zurückgeben
 return theCurrentNode.evaluation;
}
```

9.2 Der $TD - prob^n(\lambda)$-Lernalgorithmus

```
// temporalle Differenz aus Sicht des Spielers zum Zeitpunkt t ermitteln
public void computeError(IState theEndState, int t) {

 // Anzahl aller in der Partie gespielten Züge
 int N = theEndState.getNumPerformedMoves();

 // Id des untersuchenden Spielers
 int ownId = thePlayer.getPlayersPosition();

 // der Erwartungswert aus Sicht der Suchenden Spielers zum Zeitpunkt t
 IStateActionVariant[] allSubVariant = theEndState.getStoredVariant(t).getVariants();
 float[] currentEvaluation = theEndState.getStoredVariant(t).getEvaluation();
 int nSubVariants = allSubVariant.length;

 allErrors[t] = new lambdaJump(nSubVariants);

 // der Laufindex des jeweils nächsten Zuges und des jeweiligen Spielers
```

```
 int moveIndex = t+1;
 int currentPlayerPosition = ownId;

 // den nächsten eigenen Zug ermitteln
 if(moveIndex < N) {

  // die Id des Spielers zum nächsten Zeitpunkt ermitteln
  currentPlayerPosition = theEndState.getPlayerOfMove(moveIndex);

  // gegnerische Züge bis zum nächsten eigenen Zug überpringen
  while(ownId != currentPlayerPosition && moveIndex<(N-1)) {

   // Laufindex erhöhen
   moveIndex++;
   currentPlayerPosition = theEndState.getPlayerOfMove(moveIndex);
  }
 }

 // die Erwartung des Spielers zum nächsten eigenen Zeitpunkt (evtl Spielende)
 float[] nextResult;

 // kam der Spieler nicht noch einmal dran?
 if(moveIndex >= N || currentPlayerPosition != ownId) {

  // dann entspricht die Erwartung dem realen Spielausgang
  int[] gameResult  = thePlayer.getObservingGame().getGameResult();
  nextResult = new float[gameResult.length];

  // Werte des Spielausgangs übernehmen
  for (int i = 0; i < gameResult.length; i++) {
   nextResult[i] = (float)gameResult[i];
  }

 // sonst die Erwartung zum nächsten eigenen Zeitpunkt übernehmen
 } else {
  nextResult = theEndState.getStoredVariant(moveIndex).getEvaluation();
 }

 // temporale Differenz und nächsten eigenen Index abspeichern
 allErrors[t].next_t = moveIndex;
 for(int i=0; i<nSubVariants; i++) {

  // temporale Differenz zwischen Entscheidungsvariante
  // und nächstem eigenen Index
  allErrors[t].variantErrors[i] = nextResult[ownId]
                            - allSubVariant[i].getEvaluation()[ownId];
 }

 // temporale Differenz zwischen Gesamtvariante und nächstem eigenen Index
 allErrors[t].totalError = nextResult[ownId] - currentEvaluation[ownId];

 return;
}

// Koeffizienten lernen aus Sicht des Spielers theCurrentPlayer
```

```
// für die gespielte Partie theEndState
public boolean trainCoefficients(IState theEndState,
                                 IPlayer theCurrentPlayer) {

 // den Lernenden Spieler speichern
 setPlayer(theCurrentPlayer);

 // die Bewertungsklasse des Spielers
 IEvaluation theEval = thePlayer.getEvaluation();

 // nur wenn Koeffizienten vorliegen kann trainiert werden
 if(theEval.getNumStateClasses() > 0 ) {

  // Anzahl aller in der Partie gespielten Züge
  int N = theEndState.getNumPerformedMoves();

  // Speicherobjekte für die temporalen Differenzen und die Zuguindizes
  allErrors = new lambdaJump[N];

  // die Id des lernenden Spielers ermitteln
  int ownId = thePlayer.getPlayersPosition();

  // alle Züge durchgehen
  for (int i = 0; i < N; i++) {

   // und nach eigenen Züge suchen
   int idHere = theEndState.getPlayerOfMove(i);
   if(idHere == ownId) {

    // den temporalen Differenzvektor d_t^i berechnen
    computeError(theEndState, i);
   }
  }

  // den ursprünglichen Koeffizientenvektor sichern
  float[] originalVec = theEval.getClassCoefficients(0).getClassCoefficientsVector();

  // eine Kopie des Koeffizientenvektors erstellen
  float[] updapteVec  = new float[originalVec.length];

  // die Partie zurückdrehen
  IStateAction[] allPerformedMoves = theEndState.getActionHistory();
  for(int t=0; t<N; t++) {
   theEndState.undoLastAction();
  }

  // und nochmal alle eigenen Indizes durchgehen
  for(int t=0; t<N; t++) {

   // wer hat den t-ten zug ausgeführt?
   int idHere = allPerformedMoves[t].getActionerId();

   // wenn es der lernende Spieler war, dann lernen
   if(idHere == ownId) {
```

```
  // alle Entscheidungsvarianten ermitteln
  IStateActionVariantTree theCurrentVariant = theEndState.getStoredVariant(t);
  IStateActionVariant[] allVariants = theCurrentVariant.getVariants();

  // alle Varianten durchgehen
  for(int v=0; v<allVariants.length; v++) {

   int lambdaCounter = 0;
   float sum = 0;

   // Summe LAMBDA^(j-t)*d_j^v berechnen
   for(int j=t; j<N; ) {

    // Wert LAMBDA^(j-t) berechnen
    float lamdaFac = (float)Math.pow(lambda, (float)lambdaCounter);

    // Faktor d_j^v * LAMBDA^(j-t)
    float tmpSum = 0;

    // die entsprechende temporale Differenz
    if(j==t) {
     // temporale Differenz der v.ten Entscheidungsvariante
     tmpSum = allErrors[j].variantErrors[v] * lamdaFac;
    } else {
     // temporale Differenz der Stellung
     tmpSum = allErrors[j].totalError * lamdaFac;
    }
    sum += tmpSum;

    // LAMBDA ein zyklus weiter
    lambdaCounter++;

    // zur nächsten eigenen Stellung springen
    j = allErrors[j].next_t;

    // falls es keine mehr gibt -> Partie beendet
    if(j==-1)
     break;
    }

    // Ableitung berechnen für jede Entscheidungsvariante (DELTA J'(x_t, w))
    float[] ableitung = theEval.getDerrivation(theEndState, allVariants[v]);
    float factor = sum * allVariants[v].getProbability();
    updapteVec = sumVectors(updapteVec, mulVector(ableitung, factor));
   }
  }

  // Partie nach und nach erneut durchspielen
  theEndState.performAction(allPerformedMoves[t]);
 }

 // Koeffizienten aktualisieren
 theEval.getClassCoefficients(0).updateWeightVector(updapteVec);

 // er konnte gelernt werden
```

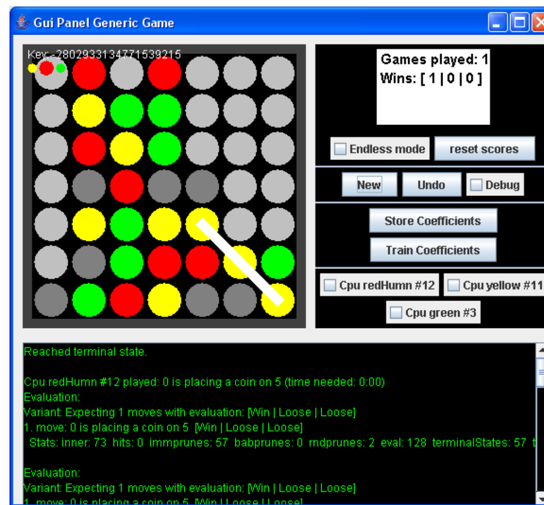


Abbildung 9.1: Ein Bildausschnitt des Spiel 3WinsProb in jGameAI.

```
      return true;
    }

    // es konnte nicht gelernt werden
    return false;
  }
```

9.3 Die Spiele 3Wins und 3WinsProb

Bei diesen Spielen handelt es sich um eine Variante von 4-gewinnt für 3 Spieler. Sie werden ebenfalls auf einem $7x7$ Gitter gsum * allVariants[v].getProbability()espielt. Dabei versuchen drei gegeneinander spielende Spieler (gelb, rot und grün) als erster durch abwechselndes Plazieren eine horizontale, vertikale oder diagonale Kombination aus 3 eigenen Steinen und damit den Sieg zu erreichen. In J3WinsProb wird zusätzlich nach jedem dritten Zug in einer zufällig gewählten Spalte s mit der Wahrscheinlichkeit $p_s = \frac{Freie\ Felder\ Spalte\ s}{Freie\ Felder\ gesamt}$ eine Niete (dunkelgrau) plaziert, die nicht für eine Kombination verwendet werden kann.

Beide Spiele eignen sich gut für das Untersuchen von Such- und Lernalgorithmen, da sie kurz und in ihrer Komplexität durch den relativ kleinen Verzweigungsfaktor von 7 gut handhabbar sind.

Literaturverzeichnis

[Allen 1989] Allen J., "*A note on the computer solution of connect-four*", in Levy D., Beal D. (Hrsg.): "*Heuristic Programming in Articial Intelligence: The First Computer Olympiad*", Ellis Horwood, 1989

[Allis 1994] Allis V.: "*Searching for Solutions in Games and Artificial Intelligence*", University of Limburg, Doktoarbeit, 1994

[Allis 1991] Allis L., van den Herik H., Herschberg I.: "*Which games will survive*", in Levy D., Beal D. (Hrsg.): "*Heuristic Programming in Articial Intelligence: The Second Computer Olympiad*" , Ellis Horwood, 1991

[Allis 1988] Allis V.: "*A Knowledge-based Approach of Connect-Four*", Vrije Universiteit Amsterdam, Masterarbeit, 1988

[Amann 1999] Amann E.: "*Evolutionäre Spieltheorie - Grundlagen und neue Ansätze*", Physica Verlag, 1999

[Baxter 2000] Baxter J., Trigdell A., Weaver L.: "*Learning to Play Chess Using Temporal Differences*", Machine Learning, 2000

[Baxter 1998] Baxter J., Trigdell A., Weaver L.: "*Knightcap: A chess program that learns by combining td(λ) with game-tree search*", Proc. Int. Conf. on Machine Learning (ICML), 1998

[Baxter 1997] Baxter J., Trigdell A., Weaver L.: "*TDLeaf(λ) Combining Temporal Difference Learning with Game-Tree Search()*",

[Billings 2006] Billings D.: "*Algorithms and Assessment in Computer Poker*", Univesiry of Alberta, Doktorarbeit, 2006

[Billings 2002] Billings D, Davidson A., Schaeffer J., Szafron D.: "*The Challenge of Poker*", Artificial Intelligence Journal, 2002

[Blair 1996] Blair J., Mutchler D., Van Lent M.: "*Perfect recall and pruning in games with imperfect information*", Computational Intelligence, 1996

[Block 2008] Block M., Bader M., Tapia E., Ramírez M., Gunnarsson K., Cuevas E., Zaldivar D., Rojas R.: "*Using Reinforcement Learning in Chess Engines*", CONCIBE SCIENCE, 2008

[Block 2005] Block M., Rauschenbach A., Buchner J., Jeschke F., Rojas R.: "*Das Schachprojekt FUSc#*", Freie Universität Berlin, Technical Report, 2005

[Block 2004] Block M.: "*Verwendung von Temporale-Differenz-Methoden im Schachmotor FUSc#*", Freie Universität Berlin, Diplomarbeit, 2004

[Breuker 1995] Breuker, D, Uiterwijk, J.: "*Transposition Tables in Computer Chess. New Approaches to Board Games*", International Institute for Asian Studies (IIAS), 1995

[Campbell 2002] Campbell M., Hoane A., Hsu F.: "*Deep Blue*", Artificial Intelligence, 2002

[Catan (URL)] http://www.catan.com

[c't contest (URL)] http://www.heise.de/ct/creativ/08/02/

[Czerwionka 2008] Czerwionka P., Block M., Bader M., Rojas R.: "*Maschinelles Spielen im Klassiker Asteroids*", Freie Universität Berlin, Technical Report, 2008

[Donninger 2004] Donninger C., Lorenz U.: "*The Chess Monster Hydra*", Proc. Int. Conf. on Field-Programmable Logic and Applications (FPL), 2004

[Ertel 2008] Ertel W.: "*Grundkurs Künstliche Intelligenz : Eine praxisorientierte Einführung*", vieweg-Verlag, 2008

[Feinstein 1993] Feinstein J.: "*Amenor wins world 6×6 championships!*", British Othello Federation Newsletter, 1993

[Gasser 1996] Gasser R.: "*Solving Nine Men's Morris*", in Nowakowski R. (Hrsg.): "*Games of No Change*", Mathematical Sciences Research Institute, 1996

[GDL (URL)] http://games.stanford.edu/language/language.html

[Ghory 2004] Ghory I.: "*Reinforcement Learning in board games*", University of Bristol, Technical report, 2004

[Ginsberg 2001] Ginsberg M.: "*Imperfect information in a computationally challenging game*", Journal Artificial Intelligence, 2001

[Goodrich 2002] Goodrich M., Tamassia R.: "*Data Structures and Algorithms in Java*", John Wiley & Sons, 2. Auflage von 2002

[Java (URL)] http://java.sun.com/javase/6/docs/api/

[jGameAI (URL)] https://sourceforge.net/projects/jgameai

[Herik 2002] van den Herik H., Uiterwijk J., van Rijswijck J.: "*Games solved: Now and in the future*", Artificial Intelligence, 2002

[Heule 2007] Heule M., Rothkrantz L.: "*Solving games - Dependence of applicable solving procedures*", Science of Computer Programming, 2007

[Korf 1991] Korf R.: "*Multiplayer Alpha-Beta Pruning*", Artificial Intelligence Vol. 48, 1991

[Knuth 1975] Knuth D., Moore R.: "*An Analysis of Alpha-Beta Pruning*". Artificial Intelligence Vol. 6, 1975

[Kuhn 1953] Kuhn H.: "*Extensive games and the problem of information*" in Kuhn H., Tucker A. (Hrsg.): "*Contributions to the Theory of Games II*", Princeton University Press, 1953.

[Land 1960] Land A., Doig A.: "*An automatic method for solving discrete programming problems*", Econometrica, 1960

[Luce 1957] Luce D., Raffia H. "*Games and Decisions*". Dover Publications, 1985

[Luger 2001] Luger G.: "*Künstliche Intelligenz - Strategien zur Lösung komplexer Probleme*", Pearson Studium, 2001

[Luckhardt 1986] Luckhardt C., Irani K.: "*An algorithmic solution of N-person games*", Proc. Int. Conf. on Artificial Intelligence (AAAI), 1986

[Marsland 1986] Marsland A.: "*A review of game-tree pruning*" J. Int. Computer Chess Assoc., 1986

[Mitchell 1997] Mitchell T.: "*Machine Learning*", McGtaw-Hill, 1997

[Nash 1950] Nash J.: "*Non-cooperative Games*", Princeton University, Doktorarbeit, 1950

[Neumann 1928] von Neumann J.: "*Zur Theorie der Gesellschaftsspiele*", in Mathematische Annalen, 1928

[Peterson 2002] Peterson G., Reif J.: "*Decision algorithms for multiplayer non-cooperative games of incomplete information*", Journal of Computers and Mathematics, 2002

[Poker Academy (URL)] http://www.poker-academy.com/

[Polaris (URL)] http://poker.cs.ualberta.ca/

[Rich 1983] Rich E.: "*Artificial Intelligence*", McGtaw-Hill, 1983

[Rieck 2007] Rieck C.: "*Spieltheorie - Eine Einführung*", Christian Rieck Verlag, 7. Auflage, 2007

[Rijswijck 2003] van Rijswijck J., Hayward R., Björnsson Y., Johanson M., Kan M., Po N.: "*Solving 7x7 Hex: Virtual Connections and Game State Reduction Advances*", Computer Games, 2003

[Rojas 2002] Rojas R., Göktekin C., Friedland G., Krüger M., Scharf L.: "*Konrad Zuses Plankalkül – Seine Genese und eine moderne Implementierung*", Freie Universität Berlin, 2002

[Rojas 1993] Rojas R.: "*Theorie der neuronalen Netze*", Springer-Verlag, 1993

[Romein 2002] Romein J., Bal H.: "*Awari is Solved*", Journal of the ICGA, 2002

[Russel/Norvig 2004] Russel S., Norvig P.: "*Künstliche Intelligenz*", 2. Auflage, Pearson Studium, 2004

[Samuel 1959] Samuel A. L.: *"Some Studies in Machine Learning Using the Game of Checkers"* IBM Journal of Research and Development, 1959

[Schaeffer 2007] Schaeffer J., Burch N., Björnsson Y., Kishimoto A., Müller M., Lake R., Lu P., Sutphen S.: "*Checkers Is Solved*", Science, 2007

[Schaeffer 1996] Schaeffer J., Lake R.: "*Solving the Game of Checkers*", Mathematical Sciences Research Institute, 1996

[Schaeffer 1989] Schaeffer J.: *"The History of Heuristic and Alpha-Beta Search Enhancements in Practice"*, IEEE Transactions on Pattern Analysis and Machine Learning, 1989

[Schauenberg 2006] Schauenberg T.: *"Opponent Modelling and Search in Poker*", University of Alberta, 2006

[Schraudolph 1994] Schraudolph N., Dayan P., Sejnowski T.: "*Temporal Difference Learning of Position Evaluation in the Game of Go*" in Cowan, J., Tesauro, G., Alspector, J. (Hrsg.): "*Advances in Neural Information Processing Systems 6*", Morgan Kaufmann, 1994

[Shannon 1963] Shannon C., Weaver W.: "*The Mathematical Theory of Communication*", University of Illinois Press, 1963

[Shannon 1950] Shannon, C.: "*Programming a Computer for Playing Chess*", Philosophical Magazine Ser.7, Vol. 41, 1950

[Spiele AG (URL)] http://page.mi.fu-berlin.de/block/gameinggroup/startseite.html

[Standford (URL)] http://games.stanford.edu/

[Steinwender 1995] Steinwender D., Friedel F.: "*Schach am PC - Bits und Bytes im königlichen Spiel*", Markt & Technik, 1995

[Sturtevant 2006-1] Sturtevant N., Bowling M.H.: "*Robust game play against unknown opponents*", Proc. Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS), 2006

[Sturtevant 2006-2] Sturtevant N., Bowling M., Zinkevich M.: "*ProbMaxN : Opponent Modeling in N-Player Games*", Proc. Int. Conf. on Artificial Intelligence (AAAI), 2006

[Sturtevant 2006-3] Sturtevant N., White A.: "*Feature Construction for Reinforcement Learning in Hearts*", Computers and Games, 2006

[Sturtevant 2004] Sturtevant N.: "*Current Challenges in Multi-Player Game Search*", Computers and Games 2004, Springer, 2004

[Sturtevant 2003-1] Surtevant N.: "*Last-Branch and Speculative Pruning Algorithms for* Max^n", International Joint Conference on Artificial Intelligence (IJCAI), 2003

[Sturtevant 2003-2] Surtevant N.: "*Multi-Player Games Algorithms and Approaches*", University of California, Los Angeles, Doktorarbeit, 2003

[Sturtevant 2002] Sturtevant N.: "*A Comparison of Algorithms for Multi-Player Games*", Computers and Games, 2002

[Sturtevant 2000] Sturtevant N, Korf R.E.: "*On pruning techniques for multi-player games*", Proc. Int. Conf. on Artificial Intelligence (AAAI), 2000

[Sutton 1998] Sutton R., Barto A.: "*Reinforcement Learning: An Introduction*", MIT-Press, 1998

[Sutton 1990] Sutton, R.S., Barto A.G.: "*Time Derivative Models of Pavlovian Reinforcement*", Learning and Computational Neuroscience, 1990

[Tesauro 1995] Tesauro G.: "*Temporal difference learning and td-gammon*", Communications of the Association for Computing Machinery (CACM), 1995

[Thomas 2003] Thomas R.: "*Real-time decision making for adversarial environments using a plan-based heuristic*", University of EVANSTON, 2003

[Thomas 2002] Thomas R., Hammond K.: "*Java settlers: a research environment for studying multi-agent negotiation*", Proc. Int. Conf. on Intelligent User Interfaces (UIU), 2002

[Turing 1953] Turing A.: „*Chess*“, in "*Faster than Thought*", Pitman, London, 1953

[Wang/M 2008] Wang M., Bader M., Block M., Rojas R.: "*Intelligent Agent-Based Board Games for Multi-Touch Systems*", Microsoft Academic Days (MSAD), 2008

[Wang/M 2008-2] Wang M.: "*Intelligente agentenbasierte Spielsysteme für intuitive Multi-Touch-Umgebungen*". Freie Universität Berlin, Diplomarbeit, 2008.

[Wang/Y 2007] Wang Y., Gelly S.: "*Modifiction of UCT with Patterns in Monte-Carlo Go*", Computational Intelligence and Games, 2007

[Warnock 1988] Warnock, T., Wendroff, B.: "*Search Tables in Computer Chess*", Journal of the International Computer Chess Association (ICCA), 1988

[Weicker 2007] Weicker K.: “*Evolutionäre Algorithmen*”, Teubner Verlag, 2007 (2. Auflage)

[Yang 2002] Yang, J., Liao, S., Pawlak, M.: “*New Winning and Losing Positions for 7x7 Hex*", Int. Conf. on Computers and Games, 2002

[Yoshizoe 2007] Yoshizoe K., Kishimoto A., Müller M.: “*Lambda Depth-First Proof Number Search and Its Application to Go*”, Int. Joint Conf. on Artificial Intelligence (IJCAI), 2007

[Zipproth 2003] Zipproth S.: “*Suchet, so werdet ihr finden*”, ComputerSchach & Spiele, 2003

[Zobrist 1970] Zobrist, A.: “*A New Hashing Method with Application for Game Playing*”, University of Wisconsin, Technical Report, 1970

Sofern nicht anders angegeben, waren alle URLs am 20. November 2008 gültig.