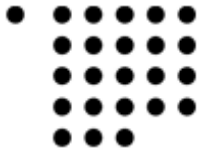

Fachhochschule Köln
Cologne University of Applied Sciences

Einsatz von N-Tupel-Systemen mit TD-Learning für strategische Brettspiele am Beispiel von Vier Gewinnt

Praxisprojekt

Fachhochschule Köln
Campus Gummersbach
Studiengang Technische Informatik
Wintersemester 2011 / 2012

Betreuer: Prof. Dr. Wolfgang Konen

Datum: 09.01.2012

Ausgearbeitet von	**Matrikelnummer**
Markus Thill	11060017

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

Die Untersuchung strategischer Brettspiele stellt *das* klassische Forschungsgebiet im Bereich der Künstlichen Intelligenz dar. In unzähligen Arbeiten wurden und werden die unterschiedlichsten Ansätze untersucht, die auch in ihren Zielsetzungen oft sehr unterschiedlich sind. Insbesondere komplexe Spiele wie *Schach* oder *Dame* scheinen aus Sicht der Forschung besonders interessant für Untersuchungen zu sein. In diesem Praxisprojekt wurde primär das Spiel *Vier Gewinnt* betrachtet, dass zwar auch eine vergleichsweise hohe Zustandsraumkomplexität aufweist, diese liegt dennoch deutlich unter der des *Schach*- bzw. *Dame*-Spiels. Zu Beginn des Projektes wurde auch das triviale Spiel Tic Tac Toe untersucht, da dieses dem Vier Gewinnt ähnlich ist und sich daher für erste grundlegende Versuche besonders eignet. In dieser Arbeit wird jedoch noch kein Bezug hierauf genommen.

Ein geeigneter Ansatz zur Erlernung von strategischen Brettspielen stellt das Reinforcement Learning – insbesondere das TD-Learning, das in diesem Projekt eingesetzt wird – dar, bei dem ein Agent versucht den Nutzen von Aktionsfolgen zu ermitteln. Nach Ablauf der Trainingsspiele erhält der Agent – je nach Ausgang Spiels – unterschiedliche Belohnungen. So kann der Nutzen einzelner Aktionsfolgen während einer bestimmten Anzahl von Trainingsspielen gewissermaßen rückwärts erlernt werden. Ein wichtiger Aspekt stellt hierbei die Modellierung der Spielfunktion dar, die im Wesentlichen zur Stellungsbewertung und letztendlich zur Bewertung einzelner Aktionen (Nutzenfunktion) benötigt wird.

Eine mögliche Approximierung der Spielfunktion mithilfe von N-Tupel-Systemen soll in Kapitel 2 beschrieben werden. Zuvor wird im nachfolgendem Abschnitt auf das Spiel Vier Gewinnt eingegangen, da sich das Projekt hauptsächlich auf dieses Spiel konzentriert. Kapitel 3 beschreibt die Besonderheiten bei der Verwendung eines N-Tupel Systems innerhalb einer TD-Learning Umgebung. Ein großer Teil der Arbeit (Kapitel 4) widmet sich der Entwicklung eines perfekt spielenden Agenten für Vier Gewinnt, der auf der Alpha-Beta Suche basiert. Dieser Agent wird vor allem als Vergleichsmöglichkeit benötigt, um beispielsweise die Spielstärke von trainierten TD-Agenten einzuordnen.

Alles in Allem soll diese Praxis-Projekt-Arbeit einen ersten Überblick geben und bereits einige der oben genannten Punkte theoretisch thematisieren. Eine Diskussion und Bewertung der Resultate wird in dieser Arbeit noch nicht vorgenommen. Diese noch fehlenden Punkte, sowie die Erläuterungen zum Stand der Technik und weitere Themen, werden zu einem späteren Zeitpunkt (Bachelor-Thesis) angesprochen.

1.1 Vier Gewinnt

Vier Gewinnt (engl. Connect Four) ist Strategiespiel für zwei Spieler, das völlig ohne Zufallselemente auskommt. Hauptziel beider Spieler ist es, vier eigene Spielsteine in eine zusammenhängende Reihe zu bringen, wodurch das Spiel gewonnen ist.

Gespielt wird *Vier Gewinnt* auf einem Brett mit sieben Spalten und sechs Zeilen. Die Besonderheit des Spiels besteht darin, dass die Spielsteine nur von oben in eine der sieben Spalten geworfen werden können, sodass die Steine in die unterste freie Zelle der entsprechenden Spalte fallen. Der Spieler der das Spiel beginnt (*anziehender Spieler*) verwendet in der Regel gelbe und der zweite Spieler (*nachziehender Spieler*) rote Spielsteine. Die Kontrahenten werfen die Spielsteine abwechselnd in die sieben Spalten, das Aussetzen eines *Halbzuges*[1] ist nicht möglich. Ist eine Spalte vollständig belegt, kann kein weiterer Halbzug in diese Spalte vorgenommen werden. Das Spiel endet daher spätestens nach 42 Spielzügen und ist in diesem Fall als Unentschieden zu werten. Gelingt es jedoch einem der beiden Spieler vorher vier eigene Spielsteine horizontal, vertikal oder diagonal in einer Reihe zu platzieren, gewinnt der Betreffende das Spiel. Bei perfekten Spiel beider Kontrahenten gewinnt immer der anziehende Spieler (Gelb). Dazu muss dieser den ersten Stein in die mittlere Spalte werfen, in allen anderen Fällen endet das Spiel mit einem Sieg des nachziehenden Spielers oder geht unentschieden aus.

[1] Um Missverständnisse zu vermeiden, wird oft (auch in dieser Arbeit) der Begriff Halbzug anstelle von Zug verwendet, vor allem dann, wenn die Bedeutung aus dem Kontext nicht klar ersichtlich ist. Der Begriff hat sich vor allem im Schach durchgesetzt. Dort ergeben zwei hintereinander ausgeführte Aktionen (jeweils eine Aktion von Weiß und Schwarz) einen Zug. Die individuelle Aktion eines Spielers wird daher Halbzug genannt (engl. ply).

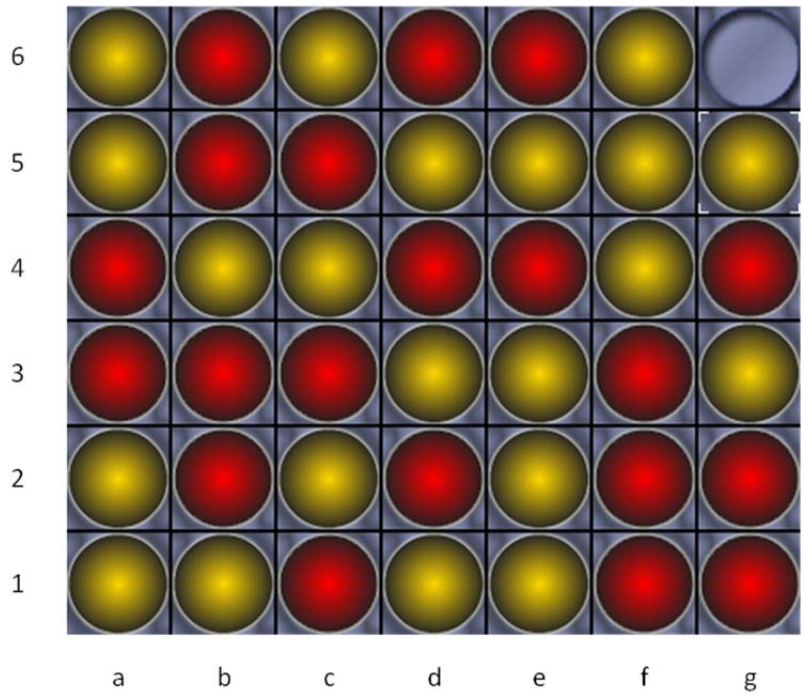

Abbildung 1.1: Typische *Vier Gewinnt* Stellung. Das Spiel wurde durch den anziehenden Spieler in seinem letzten Zug gewonnen (bei perfektem Spiel kann der Nachziehende die Niederlage bis zu seinem letzten Spielstein hinauszögern)

Starke Spieler versuchen möglichst früh Drohungen (engl. threat), also Viererketten mit einer freien Zelle (die unbelegte Zelle wird hierbei als *Drohung* bezeichnet), zu erstellen. Zum Spielende hin entscheiden dann vor allem Anzahl und Anordnung der Drohungen über den Spielausgang, da häufig einer der beiden Spieler unter *Zugzwang* gerät. Anfänger hingegen verlieren das Spiel oft frühzeitig, wenn sie *unmittelbare Drohungen* (engl. immediate threat) – also direkt zugängliche Drohungen – des Gegners übersehen.

Fälschlicherweise wird die Komplexität des Spiels zu oft unterschätzt. So ist auch heute noch ein sehr hoher Aufwand nötig, um *Vier Gewinnt* vollständig durchzurechnen. Lange Zeit war dies aufgrund der beschränkten Rechnerkapazitäten nicht möglich. So ist die vollständige Lösung des Spiels erst im Jahre 1988 gelungen [1].

Eine sehr vage Schätzung für die Größe des Zustandsraums lautet $3^{42} \approx 1{,}09 \cdot 10^{20}$ (bei drei Zuständen pro Zelle), wobei dies auch gleichzeitig eine obere Schranke für den tatsächlichen Wert darstellt. *John Tromp* ist es gelungen eine exakte Zahl an möglichen Spielzustände zu bestimmen [2], die deutlich kleiner ist und mittlerweile auch verifiziert wurde: $4531985219092 \approx 4{,}53 \cdot 10^{12}$. Dessen ungeachtet ist die Zustandsraum-Komplexität weiterhin sehr hoch, sie liegt beispielsweise noch über der Komplexität des *Mühle*-Spiels ($\approx 10^{10}$) [3].

2 N-Tupel-Systeme zur Funktionsapproximierung

Das primäre Ziel dieses Projektes ist es, einen Reinforcement-Learning Agenten zu entwickeln, der das Brettspiel "Vier-Gewinnt" erlernen kann. Zur Approximation der Spielfunktion soll ein N-Tupel System verwendet werden, dass mithilfe einer "Temporal Difference Learning"- Umgebung trainiert wird. Diese Vorgehensweise hat bereits beim Spiel Othello sehr gute Ergebnisse geliefert.

Im Folgenden soll das oben genannte Verfahren beschrieben werden, das sich insbesondere zur Abbildung der Spielfunktion (engl. value function) eignet.

2.1 Einführung

Simon M. Lucas hat im Jahre 2008 erstmalig ein neues Verfahren zur Approximation der Spielfunktion vorgestellt. Hierbei handelt es sich um sogenannte *N-Tupel Systeme*, die in diesem Kapitel beschrieben werden sollen (basierend auf [4]).

Während N-Tupel Systeme an sich bereits länger bekannt sind und in anderen Bereichen erfolgreich eingesetzt wurden, ist deren Verwendung im Zusammenhang mit lernenden Spielfunktionen neu. *Lucas* hat, laut eigenen Angaben, mithilfe des Temporal Difference Learnings (siehe Kapitel 3) ein N-Tupel System für das Othello-Spiel trainiert, das den bisherigen linearen und künstlichen neuronalen Netzen überlegen ist. So konnten die stärksten dieser Netze bereits nach 500 Selfplay-Spielen geschlagen werden[2]. Allerdings wird in dem erwähnten Artikel keine Angabe zur tatsächlichen Spielstärke, also im Vergleich zu starken Programmen wie z.B. Logistello gemacht[3].

[2] Für die Vergleiche wurden keine Baumsuchverfahren oder Ähnliches herangezogen. Lediglich die Expansion des aktuellen Zustandes und die Bewertung der Folgezustände war zulässig.

[3] Wie Lucas betont, war das primäre Ziel der Arbeit auch nicht die Erschaffung eines starken Spielers. Vielmehr stand die Lerngeschwindigkeit und die Effektivität der Lernverfahren im Vordergrund.

2.2 Die Spielfunktion

Das Ziel eines jeden Spielagenten sollte im Regelfall sein, während eines Spiels Züge zu machen, die ihm langfristig den größtmöglichen Vorteil verschaffen, bestenfalls eine Gewinnposition. Aus diesem Ziel folgt unmittelbar, dass der Agent für jeden Spielzustand die Qualität der möglichen legalen Spielzüge einschätzen muss und anschließend, den für sich besten heraussucht und ausführt.

Zur Beurteilung könnten beispielsweise alle legalen Züge sukzessive ausgeführt und der Wert für die Folgezustände berechnet werden. Dies geschieht mithilfe der sogenannten *Spielfunktion* $V(s_t)$, die als Argument einen Spielzustand $s_t \in S$ zu einem bestimmten Zeitpunkt erhält und diesen auf eine reellen Wert $V(s_t) \in \mathbb{R}$ abbildet, wobei S die Menge aller möglichen legalen Spielzustände beschreibt. Häufig ist der Funktionswert auf ein bestimmtes Intervall beschränkt (z.B. $V(s_t) \in [-1, +1]$). Im Allgemeinen sind niedrige Werte für den einen Spieler vorteilhaft, hohe dagegen für dessen Gegner (vgl. [5]).

Eine perfekte Spielfunktion hätte man dann, wenn jedem Spielzustand s_t der korrekte Wert zugeordnet wird. Solch eine Spielfunktion könnte man etwa mithilfe einer Tabelle realisieren, die für jeden Zustand einen entsprechenden Eintrag enthält. Während ein derartiger Ansatz für Spiele mit einem kleinen Zustandsraum $|S|$ (z.B. *Tic Tac Toe*) noch denkbar ist, wäre der Speicherbedarf für komplexere Spiele wie *Vier Gewinnt* beträchtlich. Aus diesem Grund geht man in solchen Fällen dazu über, die Spielfunktion mithilfe einer speziellen Funktion zu approximieren. Häufig kommen hierzu lineare Funktionen oder künstliche neuronale Netze zum Einsatz, die mithilfe des Reinforcement Learnings trainiert werden.

2.3 N-Tupel Systeme

Einem durchschnittlichen menschlichen Spieler ist es nahezu unmöglich eine Vier Gewinnt Stellung als Ganzes zu untersuchen und daraus die richtigen Schlüsse zu ziehen. Vielmehr wird ein Mensch versuchen einzelne Merkmale und Muster auf dem Spielfeld zu erkennen und diese zu einem Gesamtbild zusammenzusetzen. So sind beim Spiel *Vier Gewinnt* insbesondere die Anzahl und die Anordnung der einzelnen Drohungen für den Spielausgang entscheidend. Daraus ergeben sich verschiedene Strategien, die beide Spieler verfolgen müssen.

Mithilfe von *N-Tupel Systemen* versucht man genau diesen obigen Sachverhalt umzusetzen. Es werden jeweils mehrere lokale *Ausschnitte* des Spielfeldes untersucht und deren Teilresultate zu einem *Gesamtergebnis* kombiniert. Hierzu benötigt der Entwickler keine besonderen Kenntnisse des betrachteten Spiels, er muss sich also beispielsweise keine komplexen Eingabekodierungen für neuronale Netze oder Ähnliches überlegen; dies wird bereits vom N-Tupel System übernommen, das mit wenigen Eingabegrößen auskommt und daraus einen komplexen *"Feature-Raum"* erzeugt.

Ein N-Tupel System besteht in aller Regel aus mehreren unterschiedlichen N-Tupeln, wobei einzelne N-Tupel endliche Listen der Form $\mathrm{T} = (\tau_0, \tau_1, \dots, \tau_{N-1})$ sind und jeweils eine Teilmenge aller *Abtastpunkte* P (engl. sampling points) enthalten ($\{\tau_0, \tau_1, \dots, \tau_{N-1}\} \subseteq P$). Für klassische Brettspiele entspricht jeder Abtastpunkt einer Spielfeldzelle. Das Spiel *Tic Tac Toe* hätte also $|P| = 9$ und *Vier Gewinnt* $|P| = 42$ mögliche Abtastpunkte, wobei nicht in jedem N-Tupel System alle Spielfeldzellen zwangsläufig abgetastet werden. Die *Länge* N eines N-Tupels könnte sich für beide Spiele demnach im Bereich von $1 \leq N \leq 9$ bzw. $1 \leq N \leq 42$ bewegen[4].

Je nach Spiel ist die Belegung einer Spielfeldzelle des Spielfeldes mit mehreren unterschiedlichen Figuren möglich, ein einzelner Abtastpunkt p_j kann demnach m unterschiedliche Werte annehmen. Für jede der m möglichen Belegungen ordnet man dem entsprechenden Abtastpunkt eine Zahl $p_j \in \{0,1, \dots, m-1\}$ zu, wobei $p_j = 0$ z.B. eine leere Zelle repräsentieren könnte.

Nach welchen Kriterien die Abtastpunkte ausgewählt werden ist nicht vorgeschrieben. So kann man beispielsweise für jedes N-Tupel eine gewisse Zahl an zufälligen Punkten auswählen oder zusammenhängende Ketten erzeugen. Wie sich herausstellte, eignen sich insbesondere für *Vier Gewinnt* und *Tic Tac Toe* sogenannte "*Random Walks*", also zufällige zusammenhängende Ketten, die man ausgehend von einem zufälligen Startpunkt aus generiert. Auch *Lucas* verwendet diese Generierungs-Vorschrift für sein Othello-Programm. Je nach Spiel können auch andere Arten von N-Tupeln gute Ergebnisse liefern.

[4] Wie wir später sehen werden, ist die minimale sowie maximal mögliche Länge für die meisten Brettspiele nicht empfehlenswert bzw. nicht realisierbar.

Nun ist es so, dass jedes N-Tupel T eine sogenannte *Lookup-Tabelle* (*LUT*) – im einfachsten Fall ein gewöhnliches Array – zugeordnet bekommt. Anhand der aktuellen Belegung der einzelnen Abtastpunkte eines N-Tupels berechnet man einen Tabellenindex, mit dem ein einzelnes Element in der zugehörigen LUT angesprochen wird. Welches Element angesprochen wird, hängt also von dem derzeitigem Zustand des N-Tupels und somit direkt vom Zustand des Spielfeldes ab.

Die Größe einer LUT ergibt sich aus der Länge des N-Tupels und der Anzahl an möglichen Zuständen pro Abtastpunkt:

$$|LUT| = m^N \tag{2.1}$$

Bei Spielen mit vielen Figuren, kann die LUT daher schnell sehr groß werden, sodass man die Tupellängen auf kleinere Werte beschränken muss.

Eine *Indexierungsfunktion* für ein N-Tupel lässt sich folgendermaßen beschreiben[5]:

$$\xi(\mathrm{T}) = \sum_{i=0}^{N-1} \tau_i \cdot m^i \tag{2.2}$$

Die obige Formel erzeugt eine natürliche Zahl (inklusive der Null) der Länge N im Stellenwertsystem der Basis m. Die Reihenfolge der Abtastpunkte innerhalb eines N-Tupels ist unerheblich, solange diese – einmal festgelegt – im Weiteren beibehalten wird. Dies ist vor allem für die Indexberechnung innerhalb der einzelnen LUTs wichtig.

Nach diesen Vorüberlegungen ist es nun möglich eine lineare Spielfunktion mithilfe von N-Tupel Systemen zu definieren. Betrachtet man die Elemente der Lookup-Tabellen als einfache *Gewichte*, lässt sich der Wert der Spielfunktion – abhängig vom aktuellen Spielzustand – als Summe einzelner Gewichte darstellen. Hierzu addiert man schlicht die Gewichte aller N-Tupel, die durch die Indexierungsfunktion adressiert werden. Mithilfe einer Funktion $\Theta(\mathrm{s_t}) = \{\mathrm{T}_0, \mathrm{T}_1, \dots\}_t$ bestimmt man zunächst die Werte aller nötigen Abtastpunkte P und erzeugt daraus anschließend die einzelnen N-Tupel. Die Spielfunktion lässt sich dann wie folgt beschreiben:

$$V(\mathrm{s_t}) = \sum_{\mathrm{T} \in \Theta(\mathrm{s_t})} LUT_{\mathrm{T}}[\xi(\mathrm{T})] \tag{2.3}$$

[5] In der Praxis bietet es sich an, zur Berechnung der Indexes das *Horner-Schema* heranzuziehen; dies sollte etwas Rechenzeit einsparen.

Häufig wird auch eine weitere Funktion verwendet, die den Wertebereich der Spielfunktion auf ein gewisses Intervall beschränkt. Solch eine Funktion stellt beispielsweise der *Tangens Hyperbolicus* dar, mit der die Spielfunktion folgende Form erhält:

$$\bar{V}(\mathrm{s_t}) = \tanh\big(V(\mathrm{s_t})\big) = \tanh\left(\sum_{\mathrm{T}\in\Theta(\mathrm{s_t})} LUT_{\mathrm{T}}[\xi(\mathrm{T})]\right) \tag{2.4}$$

Ein Hauptvorteil von N-Tupel-Systemen im Vergleich zu anderen liegt darin, dass es ein sehr gutes Laufzeitverhalten besitzt. Der Rechenaufwand für die Indexberechnung verhält sich logarithmisch zur LUT-Größe und linear zur Anzahl der N-Tupel. Weiterhin werden nur wenige Gewichte benötigt, um die Ausgabe der Spielfunktion zu bestimmen.

2.4 Ausnutzung von Spielfeldsymmetrien

Vielfach sind unterschiedliche Spielstellungen aufgrund von *Symmetrien* (Spiegel- und Rotationssymmetrien) identisch. In den Spielen *Tic Tac Toe* oder *Othello* können beispielsweise bis zu acht unterschiedliche Stellungen aufgrund von Spiegelungen oder Rotationen gleich sein, beim Mühle-Spiel sind sogar noch mehr Symmetrien möglich (Vertauschen von innerem und äußerem Ring.). Angesichts der Schwerkrafts-Charakteristik des Vier Gewinnt Spiels, führen hier lediglich Spiegelungen an der mittleren Achse zu äquivalenten Stellungen.

Eine Spielfunktion sollte im Idealfall daher für solch äquivalente Spielzustände das gleiche Resultat liefern. Dieser Sachverhalt kann für das Training der Spielfunktion und für die Stellungsbewertung ausgenutzt werden.

So erreicht man deutlich schneller erste Trainingserfolge, da viele weitere Situationen gelernt werden können, die das Training nie oder nur selten erreicht. Um die symmetrischen Stellungen für die Berechnung Spielfunktion mit einzubeziehen, ist es zweckmäßig, die Summe über die Ausgabe der Spielfunktion für alle symmetrischen Stellungen vorzunehmen:

$$V_{Sym}(\mathrm{s_t}) = \sum_{x \in \mathrm{SYM}(\mathrm{s_t})} V(x) \tag{2.5}$$

wobei $\mathrm{SYM}(\mathrm{s_t})$ die Menge aller äquivalenten symmetrischen Stellungen zu einem einzelnen Spielzustand $\mathrm{s_t}$ beschreibt.

Ein etwas anderer Ansatz könnte folgendermaßen aussehen: Beim Erzeugen *eines* N-Tupels generiert man parallel dazu weitere N-Tupel, die jedoch *die* Abtastpunkte enthalten, die aufgrund von Rotationen und Spiegelungen äquivalent sind. Dieser Gruppe von N-Tupeln wird anschließend die entsprechende Lookup-Tabelle zugeordnet.

2.5 Fehlende Spielerinformation im N-Tupel System

Beim Einsatz von N-Tupel Systemen kann ein Problem aufgrund der Tatsache auftreten, dass für jedes N-Tupel lediglich Ausschnitte des Spielzustandes erfasst werden. In gewisser Weise ist dies auch das Ziel eines N-Tupel Systems, man möchte ja schließlich das *Gesamtbild* in kleinere *Bruchstücke* zerlegen, um in diesen nach gewissen Mustern und Merkmalen zu suchen.

Probleme können jedoch unter Umständen dann auftreten, wenn der Spieler am Zug nicht mit codiert wird.

Die einzelnen N-Tupel enthalten lediglich eine Folge von Abtastpunkten, aus denen man den aktuellen Spieler im Regelfall nicht wieder rekonstruieren kann. Eine Ausnahme stellt hierbei ein N-Tupel maximaler Länge dar, da es das komplette Spielfeld enthält[6]. Solch ein N-Tupel widerspricht jedoch dem Grundgedanken der hinter solch einem N-Tupel System steht und ist in der Praxis – aufgrund des hohen Speicherbedarfs – längst nicht immer realisierbar. Im Regelfall lässt sich der aktuelle Spieler daher nicht ohne weiteres aus einem N-Tupel extrahieren, sodass für zwei Identische N-Tupel-Folgen auf denselben LUT-Eintrag zugegriffen wird, obwohl sich die beiden zugrundeliegenden Spielzustände hinsichtlich des aktuellen Spielers möglicherweise unterscheiden.

Diese Problematik muss allerdings nicht für alle Spiele zwangsläufig von Nachteil sein. So verwendet *Lucas* in seinem N-Tupel System für Othello offenbar keine weiteren Mechanismen, die zwischen den beiden Spielern unterscheiden können und erzielt damit dennoch sehr gute Ergebnisse. Andererseits wurden mit dieser Variante insbesondere beim *Vier Gewinnt* Spiel keine befriedigenden Resultate erreicht.

Abbildung 2.1 enthält eine *Vier Gewinnt* Stellung mit einer unmittelbaren Drohung für den anziehenden Spieler. Da Spieler Rot am Zug ist, kann dieser die Drohung in seinem nächsten Halbzug neutralisieren und das Spiel würde regulär fortgesetzt; bei perfektem Spiel beider Kontrahenten endet das Spiel letztendlich Unentschieden. Das gekennzeichnete 4-Tupel enthält jedoch nur *die* drei Steine, die die Drohung erzeugen und ansonsten keine weiteren Informationen. Es ist völlig unerheblich welcher Spieler am Zug ist, für diese Kombination wird immer der gleiche Eintrag in der zugeordneten Lookup-Tabelle angesprochen. Sollte beispielsweise Spieler Rot einen Stein in die a-Spalte werfen, ändert sich die Belegung des 4-Tupels nicht. Spieler Gelb würde das Spiel anschließend im nächsten Halbzug gewinnen. Man kann also keine Aussage darüber machen, wie diese Konstellation des 4-Tupels letztendlich zu bewerten ist, da sich – abhängig vom aktuellen Spieler – unterschiedliche Spielresultate ergeben.

[6] Mit der Einschränkung, dass der Spieler am Zug eindeutig aus der aktuellen Position bestimmbar sein muss. Bei *Vier Gewinnt* oder *Tic Tac Toe* ist das der Fall, beim *Schachspiel* allerdings nicht.

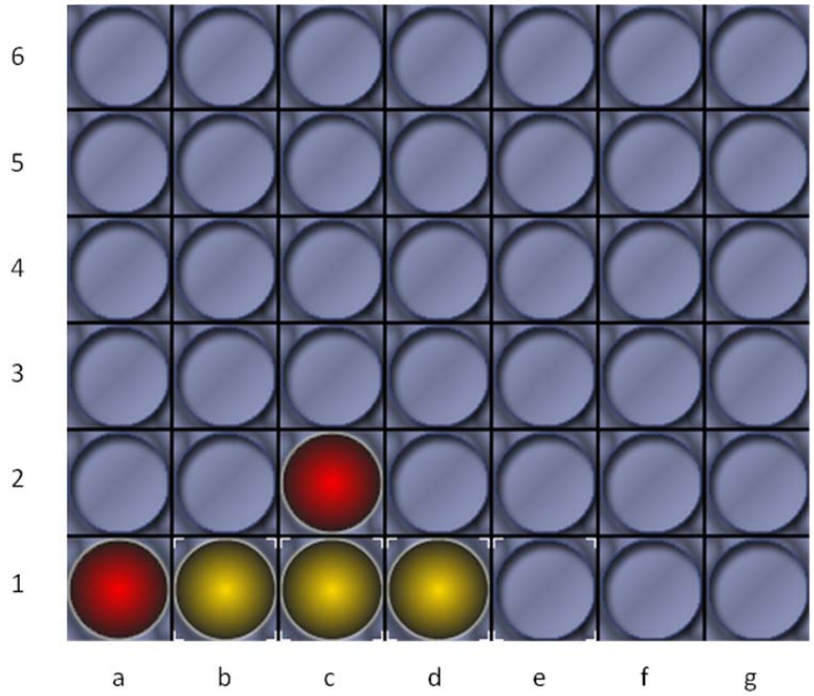

Abbildung 2.1: Stellung mit einer unmittelbaren Drohung für den anziehenden Spieler. Das betrachtete 4-Tupel wird durch die weißen Markierungen in den jeweiligen Spielfeldecken gekennzeichnet.

Die einfachste Möglichkeit, um dieser Problematik zu begegnen ist folgende: Für jedes N-Tupel legt man zwei Lookup-Tabellen an, eine je Spieler. So kann – je nach Spielzustand – auf die entsprechende Tabelle zugegriffen werden. Allerdings sollte man beachten, dass sich dadurch der Speicherbedarf verdoppelt. Dennoch liefert die saubere Trennung in jeweils zwei Lookup-Tabellen deutlich bessere Ergebnisse, als die herkömmliche Methode. Die Ergebnisse hierzu werden in dieser Arbeit jedoch noch nicht diskutiert.

2.6 Erzeugung von Eingabevektoren für lineare Netze

Die Funktionsweise eines N-Tupel-Systems mit Lookup-Tabellen entspricht im Wesentlichen dem eines linearen Netzes, das die Ausgabe mithilfe eines Gewichts- und Feature-Vektors berechnet. Im Programm selbst sollte jedoch auf die Verwendung von Vektoren verzichtet werden, da diese unnötig Ressourcen (Speicher sowie Rechenzeit) beanspruchen; vor allem, wenn viele und lange N-Tupel Verwendung finden. Beim Einsatz des TD(λ)-Algorithmus' (siehe Kapitel 3) mit $\lambda \neq 0$ könnte der Einsatz von Vektoren jedoch nötig werden, sodass sich die Verwendung eines lineares Netzes anbietet. Sollte daher solch ein lineares Netz eingesetzt werden, verzichtet man auf die Lookup-Tabellen des N-Tupel Systems. Stattdessen legt man ein lineares Netz mit einer gewissen Zahl an Gewichten an und ermittelt für jeden Spielzustand einen geeigneten Feature-Vektor. Um den Wert der Spielfunktion zu berechnen ist es dann vollkommen ausreichend, wenn man dem linearen Netz einen Feature-Vektor übergibt.

Zunächst ist die Anzahl der Gewichte des Netzes zu ermitteln, aus der sich gleichzeitig die Länge des Feature-Vektors ergibt:

$$c = \sum_{\mathrm{i}} m^{N_i} \tag{2.6}$$

Ein Vektor, der alle c Gewichte enthält, hat die Form:

$$w = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{c-1} \end{pmatrix} \tag{2.7}$$

Um die weiter oben diskutierte Indexierungsfunktion $\xi(\mathrm{T})$ weiterhin verwenden zu können, bietet es sich an, die Struktur der Lookup-Tabellen auf den Gewichts-Vektor zu übertragen. Damit die neue Indexierungsfunktion auch weiterhin eine bijektive Abbildung darstellt, also jedem N-Tupel einen eindeutigen Index zuweist, wird zu $\xi(\mathrm{T})$ ein Offset addiert, um eine Verschiebung innerhalb des Vektors zu erreichen. Legt man die Gewichte der einzelnen N-Tupel $(\mathrm{T}_{0,}\mathrm{T}_{1,}\ldots)$ hintereinander im Vektor ab, ist die Berechnung des Indexes für ein Tupel T_{ℓ} direkt möglich:

$$\tilde{\xi}(\mathrm{T}_{\ell}) = \xi(\mathrm{T}_{\ell}) + \sigma(\ell) \tag{2.8}$$

wobei ℓ die Position – beginnend mit Null – des entsprechenden N-Tupels in dem Vektor $(\mathrm{T}_{0,}\mathrm{T}_{1,}\ldots)$ beschreibt.

Daraus ergibt sich ein Offset $\sigma(\ell)$ von:

$$\sigma(\ell) = \begin{cases} 0, & f\ddot{u}r\ \ell = 0 \\ \sum_{i=0}^{\ell-1} m^{N_i}, & sonst \end{cases} \tag{2.9}$$

Im Prinzip entspricht die gewählte Darstellung einer direkten Hintereinanderreihung aller LUTs vom ursprünglichen N-Tupel System.

Mit diesen Informationen ist es nun möglich, einen Feature-Vektor $g(\mathrm{s_t})$ des aktuellen Spielzustandes zu erzeugen, der auf den einzelnen N-Tupeln basiert. Hierzu legt man zunächst einen Nullvektor der Länge c an. Abhängig vom aktuellen Spielzustand werden anschließend alle nötigen N-Tupel, wie weiter oben bereits beschrieben, mit $\Theta(\mathrm{s_t}) = \{\mathrm{T_{0,}T_{1,}} \dots\}_t$ berechnet und die einzelnen Elemente des Feature-Vektors nach folgender Vorschrift aktualisiert:

$$g_i(\mathrm{s_t}) \leftarrow g_i(\mathrm{s_t}) + 1 \quad \forall\, i \in \{\tilde{\xi}(\mathrm{T_0}), \tilde{\xi}(\mathrm{T_1}), \dots\}^7 \tag{2.10}$$

Auch die N-Tupel $\Theta\big(\mathrm{SYM}(\mathrm{s_t})\big)$ der symmetrischen Spielstellungen lassen sich ohne Weiteres mit einbeziehen (vgl. Abschnitt 2.4). Mithilfe dieses Feature-Vektors kann die Ausgabe des linearen Netzes anschließend sehr leicht berechnet werden:

$$V(\mathrm{s_t}) = f\big(w, g(\mathrm{s_t})\big) = w \cdot g(\mathrm{s_t}) \tag{2.11}$$

Auch hier ist die Verwendung einer Aktivierungsfunktion wie dem Tangens-Hyperbolicus denkbar. Die Spielfunktion in Gleichung 2.11 liefert die gleichen Ergebnisse, wie das zuvor beschriebene N-Tupel System mit mehreren Lookup-Tabellen, der nötige (Rechen-) Aufwand ist jedoch deutlich größer. Möglichweise kann der beschriebene Feature-Vektor auch als Eingabe anderer Netztypen dienen; in wie weit dies sinnvoll ist, muss im Einzelfall geprüft werden.

Der Feature-Vektor, der mithilfe des N-Tupel Systems erzeugt wird, ist nie ein Null-Vektor, da für jedes N-Tupel genau ein Element des Vektors angepasst wird (bei der Verwendung von Symmetrien auch mehrere Elemente). Die Verwendung eines BIAS-Wertes, also eines unabhängigen Gewichtes, sollte daher nicht nötig sein.

[7] Die Schreibweise $g_i(\mathrm{s_t}) = 1$ ist unter gewissen Umständen auch denkbar. Bei Ausnutzung von Spielfeldsymmetrien, kann ein einzelner Index jedoch mehrmals auftreten, sodass $g_i(\mathrm{s_t}) = 1$ nicht in jedem Fall korrekt wäre.

3 TD-Learning mit einem N-Tupel-System

Im vorherigen Kapitel wurde eine mögliche Approximierung der Spielfunktion mithilfe von N-Tupel Systemen beschrieben. Ein Verfahren, das die Spielfunktion trainiert, ist allerdings noch nicht diskutiert worden. Ein besonders geeigneter Ansatz stellt das *Temporal Difference Learning* dar, mit dessen Hilfe *Reinforcement*-Probleme gelöst werden können.

Ein Problem, dass bei praktisch allen Brettspielen auftritt, besteht darin, dass die direkte Bewertung und Einordnung einer Aktion nicht möglich ist. Erst nach einer gewissen Anzahl von Aktionen kann man eine Aussage über die Qualität der gewählten Aktionsfolge treffen; nämlich dann, wenn das Spiel beendet wird. Für den Lernprozess ist daher nicht eine einzelne Aktion entscheidend, sondern eine vollständige Sequenz derer. Eine einzelne Aktion ist erst dann als gut einzustufen, wenn diese Teil einer erfolgreichen Sequenz ist.

Probleme dieser Art können mithilfe des *Reinforcement Learnings* behandelt werden: Je nach Ausgang des Spiels erhält der Agent eine *Belohnung* oder eine *Bestrafung* (engl. *Reward*). So wird versucht, die Spielfunktion quasi rückwärts zu erlernen, indem man für jede Aktion die Bewertung des aktuellen Spielzustandes s_t etwas in die Richtung des Zielzustandes s_{t+1} anpasst. Letztendlich besteht das Ziel des Trainings also darin, mithilfe von Belohnungen und Bestrafungen eine Spielfunktion zu erlernen, um Aktionen auszuführen, die den voraussichtlichen Nutzen maximieren.

3.1 Der TD(λ)-Algorithmus

Mithilfe des in [5] und [6] beschrieben TD(λ)-Algorithmus' ist es möglich einen Agenten zu trainieren, der alleine durch *Self-Play* (Trainingsspiele gegen sich selbst) eine Spielfunktion erlernt. Weiteres Expertenwissen fließt nicht in den Trainingsprozess mit ein.

Hierzu werden eine ganze Reihe von Spielen simuliert, an deren Ende jeweils ein Reward vergeben wird. Der Algorithmus ist in leicht abgewandelter Form in Listing 3.1 aufgeführt.

Input: Anziehender Spieler p_0 [=+1 ("X") oder -1 ("O")], Initialposition s_0, sowie eine (teiltrainierte) Funktion $f(w;\ g(s_t))$ zur Berechnung der Spielfunktion $V(s_t)$

1: Setze $t \coloneqq 0$ und $e_0 \coloneqq \nabla_w f(w;\ g(s_0))$ ▷ s_0 gehört zu Spieler $-p_0$

2: **for** ($p \coloneqq p_0; s_t \notin S_{Final}\ ; p \leftarrow (-p),\ t \leftarrow t+1$) { ▷ Tausche Spieler in jedem Durchgang

3: $V_{old} \coloneqq f(w;\ g(s_t))$ ▷ w ändert sich durchgängig

4: Generiere Zufallszahl $q \in [0,1]$

5: **if** ($q < \varepsilon$)

6: Wähle After-State s_{t+1} zufällig aus ▷ Explorative Move

7: **else**

Wähle After-State s_{t+1}, der

8: $$p \cdot \begin{cases} r(s_{t+1}), & \text{für } s_{t+1} \in S_{Final} \\ f(w;\ g(s_{t+1})), & sonst \end{cases}$$ ▷ Greedy-Move (Exploitation)

maximiert

9: Bestimme Response $V(s_{t+1}) \coloneqq f(w;\ g(s_{t+1}))$ ▷ Lineares oder neuronales Netz

10: Bestimme Reward $r_{t+1} \coloneqq r(s_{t+1})$ ▷ Aus der Spielumgebung

11: Targetsignal $T_{t+1} \coloneqq \begin{cases} r_{t+1}, & \text{für } s_{t+1} \in S_{Final} \\ \gamma \cdot V(s_{t+1}), & sonst \end{cases}$ ▷ Trennung von Response und Reward i.d.R. nötig

12: Fehlersignal $\delta_t \coloneqq T_{t+1} - V_{old}$

13: **if** ($q \geq \varepsilon$ **or** $s_{t+1} \in S_{Final}$)

14: Lernschritt $w \leftarrow w + \alpha \delta_t e_t$ ▷ Für einen greedy bzw. terminierenden Halbzug

15: $e_{t+1} \coloneqq \gamma \lambda e_t + \nabla_w f(w;\ g(s_{t+1}))$ ▷ Berechne Eligibility-Traces

16: } ▷ Ende des Self-Plays

Listing 3.1: Inkrementeller TD(λ)-Algorithmus für Brettspiele

Der obige Algorithmus beschreibt genau einen Spielverlauf. Je nach Spiel können einige Hundert bis zu einige Millionen Trainingsdurchgänge notwendig werden, um erste brauchbare Resultate zu erzielen.

Nach den erfolgten Initialisierungen führt der Algorithmus für beide Spieler abwechselnd Halbzüge aus. Erst wenn ein terminierender Zustand – also eine Gewinnstellung oder ein Unentschieden – erreicht wird, ist dieser Trainingsvorgang abgeschlossen.

In jedem Durchgang benötigt man den Wert der Spielfunktion V_{old} für den aktuellen Spielzustand; hier wird das N-Tupel-System befragt. Wichtig ist, dass man hierfür nicht den Response des vorherigen Spielzustandes verwendet, sondern den Wert tatsächlich vollständig neu berechnet, da sich praktisch in jedem Schleifendurchlauf einzelne Gewichte des Systems ändern. Dies gilt insbesondere für neuronale bzw. lineare Netze, bei denen man unter Umständen doppelte Berechnungen vermeiden möchte; diese mehrfache Berechnung kann man nicht ohne Weiteres umgehen, der erhöhte Rechenaufwand muss daher in Kauf genommen werden.

Sich ständig wiederholende Spielverläufe vermeidet man, indem jeder Halbzug mit einer gewissen Wahrscheinlichkeit ε zufällig ausgewählt wird. Solche Halbzüge bezeichnet man auch als *Explorative-Moves*. Ansonsten wählt der Algorithmus für den jeweiligen Spieler einen Halbzug, der nach Befragung der Spielfunktion am vielversprechendsten erscheint (*Exploitation*).

Bei der Berechnung des *Targetsignals* T_{t+1} ist auch die Schreibweise $T_{t+1} = r_{t+1} + \gamma \cdot V(s_{t+1})$ denkbar, da der Reward für non-terminale Zustände idealerweise gleich Null ist und umgekehrt die Spielfunktion für terminale Spielzustände Null liefert. In der Praxis ist dies allerdings oft – unter anderem auch bei Verwendung eines N-Tupel-Systems – nicht der Fall, sodass eine getrennte Betrachtung nötig wird.

Ein Lernschritt findet immer dann statt, wenn ein Greedy-Zug gespielt oder ein terminaler Zustand erreicht wurde. Der Vektor e_t beschreibt hierbei die sogenannten *Eligiblity-Traces*. Wird der Parameter λ (*Trace Decay*) ungleich Null gewählt, fließen die Eligibility-Traces der vorherigen Spielzustände in den aktuellen Vektor mit ein. In dieser Arbeit soll jedoch lediglich der Fall $\lambda = 0$ betrachtet werden, sodass der Algorithmus schlichtweg den Gradienten $e_\mathrm{t} \coloneqq \nabla_\mathrm{w} f\big(w;\ g(s_t)\big)$ berechnet.

Der *Discount-Faktor* γ zinst die Werte der Spielfunktion für die Zielzustände etwas ab, da in der Regel nicht sicher ist, dass diese Zielzustände tatsächlich die bestmöglichen sind. Üblich sind Werte im Intervall von $\gamma \in [0{,}9\ ;\ 1]$.

Mithilfe der *Lernschrittweite* α vermeidet man zu starke Änderungen der Gewichte in die ein oder die andere Richtung. Typischerweise wird die Lernschrittweite im Laufe des Trainings immer weiter verkleinert, um Anfangs eine hohe Konvergenzgeschwindigkeit zu erhalten und im weiteren Verlauf stark oszillierendes Verhalten zu vermeiden. In der Praxis hat sich eine exponentielle Abnahme der Schrittweite bewährt. So sollte im Idealfall das Fehlersignal im Laufe des Trainings minimiert werden.

Letztendlich handelt es also hierbei um ein klassisches Gradientenabstiegsverfahren mit variabler Lernschrittweite, vorausgesetzt es wird $\lambda = 0$ gewählt.

3.2 Optimierung der Lernschritte

Wie weiter oben bereits erwähnt wurde, sollte man im Programm auf die Verwendung von Vektoren verzichten, sofern dies nicht unumgänglich ist. Vor allem der Feature-Vektor des N-Tupel-Systems kann unter Umständen sehr groß werden, sodass in der Praxis deutliche Laufzeitnachteile zu erwarten sind, sollte man keine geeignete Alternative finden. In Abschnitt 2.3 ist bereits ein vergleichsweise schnelles Verfahren zur Berechnung der Spielfunktion vorgestellt worden, für die einzelnen Lernschritte im TD(λ)-Algorithmus ist eine effizientere Herangehensweise noch nicht bekannt. Für den Optimierungsansatz entscheidend sind die Eligibility-Traces, die in Listing 3.1 folgendermaßen beschrieben werden:

$$e_{t+1} \coloneqq \gamma\lambda e_t + \nabla_w f\big(w;\ g(s_{t+1})\big) \tag{3.1}$$

Sollte es möglich sein, die Vektorschreibweise der Eligibility-Traces in eine etwas andere Darstellung zu überführen, ist auch eine effizientere Behandlung in der Implementierung denkbar.

Für $\lambda \neq 0$ ist eine deutliche Vereinfachung allerdings nicht ohne Weiteres vorstellbar. Da dies in der Praxis oft ohnehin nicht vorteilhaft ist, wird im Weiteren ausschließlich der Fall $\lambda = 0$ betrachtet, sodass sich obige Gleichung etwas vereinfachen lässt:

$$e_t \coloneqq \nabla_w f\big(w;\ g(s_t)\big) \tag{3.2}$$

Bei der – mithilfe des N-Tupel Systems realisierten – Spielfunktion handelt es sich um eine Funktion der Form[8]:

$$\bar{V}(s_t) = f\big(w;\ g(s_t)\big) = \tanh\big(w \cdot g(s_t)\big) \tag{3.3}$$

bzw.:

$$f\big(w;\ g(s_t)\big) = \tanh\left(\sum_k w_k \cdot g_k(s_t)\right) \tag{3.4}$$

[8] Die Ermittlung der Vektoren w und $g(s_t)$ und deren Besonderheiten sind bereits in Abschnitt 2.6 diskutiert worden.

Der Gradient lässt sich für diese Funktion vergleichsweise leicht ermitteln:

$$\begin{aligned}\nabla_{\mathrm{w}} f\big(w;\ g(s_t)\big) &= \left[1 - \tanh^2\left(\sum_k w_k \cdot g_k(\mathrm{s_t})\right)\right] \cdot g(\mathrm{s_t}) \\ &= \left[1 - f\big(w;\ g(\mathrm{s_t})\big)^2\right] \cdot g(\mathrm{s_t})\end{aligned} \tag{3.5}$$

Bei Verwendung der Aktivierungsfunktion (Tangens-Hyperbolicus) liefert der Gradient wieder den Feature-Vektor $g(\mathrm{s_t})$, der mit einem Vorfaktor multipliziert wird (Formel 3.5). Dieser Vorfaktor wird zu Eins, sollte man auf die Aktivierungsfunktion verzichten.

Nachdem der Gradient bekannt ist, lässt sich ein Lernschritt in leicht abgewandelter Form notieren:

$$w \leftarrow w + \alpha\delta_t \left[1 - f\big(w;\ g(\mathrm{s_t})\big)^2\right] \cdot g(\mathrm{s_t}) \tag{3.6}$$

Nun kann man die Tatsache ausnutzen, dass oftmals große Teile des Feature-Vektors lediglich Null-Elemente enthalten. Ein einzelnes Gewicht ändert sich während des Lernschrittes nur dann, wenn der entsprechende Eintrag im Feature-Vektor ungleich Null ist. Beim Einsatz eines N-Tupel-Systems betrifft dies ausnahmslos Einträge, die von der Indexierungsfunktion $\tilde{\xi}(\mathrm{T}_\ell) = \xi(\mathrm{T}_\ell) + \sigma(\ell)$ adressiert wurden. Es ist also vollkommen ausreichend, wenn der Wert

$$\Delta w = \alpha\delta_t \left[1 - f\big(w;\ g(\mathrm{s_t})\big)^2\right] \tag{3.7}$$

zu den – durch die Indexierungsfunktion adressierten – Gewichte addiert wird:

$$w_i \leftarrow w_i + \Delta w \cdot g_i(\mathrm{s_t}) \qquad \forall\, g_i(\mathrm{s_t}) \neq 0 \tag{3.8}$$

bzw.:

$$w_i \ \leftarrow\ w_i + \Delta w \qquad \forall\, i \in \left\{\tilde{\xi}(\mathrm{T}_0), \tilde{\xi}(\mathrm{T}_1), \ldots\right\}^9 \tag{3.9}$$

Sollte man auf ein lineares Netz mit einem individuellen Gewichts-Vektor verzichten und stattdessen die Verwendung mehrerer Lookup-Tabellen bevorzugen, wird der Gewichts-Vektor schlicht in die entsprechenden LUTs zerlegt. Formal ändert sich hierdurch jedoch nichts. Da keine Offsetberechnungen mehr nötig sind, kommt wieder die Indexierungsfunktion $\xi(\mathrm{T})$ zum Einsatz. Auch die Berechnung des Skalarproduktes $w \cdot g(\mathrm{s_t})$ zur Bestimmung der Spielfunktion kann vermieden werden, sodass man kür-

[9] Vgl. Gleichung (2.10(Analog hierzu). Bei der Ausnutzung von Spielfeldsymmetrien erweitert sich die Liste der N-Tupel um *die* der spiegel- bzw. rotationssymmetrischen Stellungen.

zere Laufzeiten erreicht, indem man die Spielfunktion nach Formel 2.3 bzw. 2.5 berechnet.

Der Übergang zu einem N-Tupel System mit getrennten Lookup-Tabellen kann folgendermaßen beschreiben werden:

$$LUT_{\mathrm{T}}[\xi(\mathrm{T})] \leftarrow LUT_{\mathrm{T}}[\xi(\mathrm{T})] + \Delta L \qquad \forall\, \mathrm{T} \in \Theta\big(\mathrm{SYM}(\mathrm{s_t})\big) \tag{3.10}$$

Analog zu Gleichung (3.7gilt:

$$\Delta L = \alpha \delta_t [1 - \bar{V}(\mathrm{s_t})^2] \tag{3.11}$$

Auch an dieser Stelle verzichtet man auf den Faktor $[1 - \bar{V}(\mathrm{s_t})^2]$, sollte die Spielfunktion $V(\mathrm{s_t})$ (keine Aktivierungsfunktion) zum Einsatz kommen.

Es ist völlig ausreichend, ΔL einmalig zu berechnen. Diesen Wert addiert man anschließend auf alle betroffenen Gewichte.

Letztendlich kann man beim Einsatz von Lookup-Tabellen vollständig auf die zeit- und speicherintensive Berechnung von Vektoren (Feature- bzw. "Lernschritts-Vektoren") während des Trainings verzichten, da nur wenige Komponenten der jeweiligen Vektoren für weitere Betrachtungen relevant wären. Dies ist insbesondere dann von Vorteil, wenn sehr viele Trainingsspiele nötig sind.

3.3 Details zur entwickelten Software

Wie auch für die – in Kapitel 4 beschriebenen – Alpha-Beta Suche, werden sogenannte Bitboards als grundlegende Datenstruktur zur Spielfeldrepräsentation des TD-Agenten eingesetzt[10]. Dies beschleunigt die Trainingsvorgänge deutlich, da viele grundlegende Operationen – beispielsweise die Erkennung von Terminalzuständen – mit wenigen CPU-Befehlen auskommen. Details hierzu können in Abschnitt 4.2 nachgelesen oder dem Quelltext (*c4.ConnectFour*) entnommen werden.

Wie bereits erwähnt wurde, erzeugt das Programm für jedes N-Tupel unabhängige Lookup-Tabellen, die mit weiteren Informationen in eigenständigen Objekten verwaltet werden. Dieser Ansatz entspricht zwar nicht ganz dem eines klassischen linearen Netzes, das alle Gewichte in einem einzelnen Vektor führt, dennoch sollten beide Systeme nach außen hin das gleiche Verhalten aufweisen. Weiterhin wird in diesem Programm auf die Erzeugung ganzer Vektoren verzichtet, da im Regelfall nur wenige Komponenten innerhalb dieser von Bedeutung sind. Ein vom N-Tupel System generierter Feature-Vektor würde hauptsächlich Null-Elemente enthalten, die für weitere Betrachtungen überflüssig wären. Man spart daher Rechenzeit und Speicher ein, wenn lediglich die wenigen relevanten Komponenten des Vektors im weiteren Verlauf heranzogen werden (siehe Abschnitt 2.6. und 3.2).

Da beim *Vier Gewinnt* Spiel die Schwerkraftsregel zum Tragen kommt, sind nicht alle leeren Felder direkt im nächsten Halbzug erreichbar. Daher kann es Sinn machen, zwischen leeren, erreichbaren Zellen und leeren, unerreichbaren Zellen zu unterscheiden, sodass jeder Abtastpunkt des N-Tupel-Systems, statt ursprünglich drei, nun theoretisch vier Zustände annehmen könnte. Allerdings würden sich dadurch die einzelnen Lookup-Tabellen deutlich vergrößern; der Speicherbedarf einer LUT für ein 8-Tupel wäre beispielsweise zehnmal größer, wenn von vier Zuständen ausgegangen wird. Das Verhältnis zwischen den beiden LUT-Größen bei einer Länge N des Tupels beträgt allgemein:

$$\frac{Größe\ der\ LUT\ für\ 3\ Zustände}{Größe\ der\ LUT\ für\ 4\ Zustände} = \left(\frac{3}{4}\right)^N \tag{3.12}$$

Ob $m = 3$ oder $m = 4$ mögliche Zustände je Zelle gewählt werden und ob die Resultate einen höheren Speicherbedarf rechtfertigen, liegt im Ermessen des Programmbenutzers.

[10] Dies gilt allerdings nur für das Vier Gewinnt Spiel. Tic Tac Toe hat eine deutlich geringere Komplexität, sodass Bitboards nicht vonnöten sind. Nichtsdestotrotz enthält die Klasse *diverses.CountPositions* eine rudimentäre Umsetzung des Bitboard-Prinzips, dass sich leicht auf den entsprechenden TD-Agenten übertragen ließe.

Weiterhin sind die Initial- bzw. Endwerte der Lernschrittweite α und der Explorationsrate ε, sowie die entsprechenden Anpassungsfunktionen über den Trainingsverlauf einstellbar. So kann unter anderem ein linearer oder exponentieller Abfall der Parameter festgelegt werden. Generell lässt sich sagen, dass niedrige Werte für die Explorationsrate das Training verlangsamen, da häufiger auf das N-Tupel-System zugegriffen wird. Vor Trainingsbeginn kann der Disount-Faktor γ auf einen konstanten Wert eingestellt werden, der Wert des Parameters λ beträgt Null und ist nicht änderbar (siehe Abschnitt 3.2).

Um die Komplexität des Vier Gewinnt Spiels zu reduzieren besteht außerdem die Möglichkeit, die einzelnen Trainingsspiele bei einer gewissen Anzahl von Spielsteinen auf dem Brett zu unterbrechen. Der Reward – daher der spieltheoretische Wert - muss anschließend von der Alpha-Beta Suche ermittelt werden. So kann man dann beispielsweise einen TD-Agenten trainieren, der die Eröffnungsphase bis zu zwölf Spielsteinen beherrscht. Vor allem lassen sich hierdurch auch die implementierten Algorithmen auf ihre Richtigkeit überprüfen.

4 Entwicklung eines Agenten für Vergleichszwecke

Um die Spielstärke des lernenden N-Tupel-Agenten bewerten zu können, sind weitere Agenten nützlich. Insbesondere wird ein Agent benötigt, der exakte Stellungsbewertungen vornehmen kann. Um dies zu erreichen ist ein Baumsuchalgorithmus notwendig. Ein perfekt spielender Agent muss den *Spielbaum* vollständig untersuchen können (im Extremfall 42 Züge), allerdings steigt der Suchaufwand exponentiell zur *Suchtiefe* an, sodass in vielerlei Hinsicht ein hoher Entwicklungsaufwand nötig ist, um dieses Ziel zu erreichen.

Dieses Kapitel soll einige der eingesetzten Verfahren und Techniken erläutern, die solch einen Agenten ermöglichen.

4.1 Die Alpha-Beta Suche

Bei der *Alpha-Beta Suche* handelt es sich um ein Baumsuchverfahren, das im Wesentlichen auf dem klassischen *Minimax-Algorithmus* beruht, jedoch in einigen Teilen optimiert wurde. Während eine einfache Minimax-Suche alle *Knoten* des Spielbaums untersucht, können bei der Alpha-Beta Suche durch die Verwendung von zwei Variablen (Alpha und Beta) unter Umständen große Teilbäume während Suche abgeschnitten werden (*Cutoff*). Anzahl und Umfang der Cutoffs hängen jedoch von einigen Faktoren wie der Güte der Zugsortierung ab.

Der Grundgedanke, der hinter dem Algorithmus steht ist folgender: Da die Opponenten aufgrund ihres maximierenden bzw. minimierenden Verhaltens gewisse Spielverläufe vermeiden wenn ihnen bereits bessere Alternativen bekannt sind, ist die Untersuchung ganzer Teilbäume unnötig, da gewisse Halbzüge ohnehin nicht in Betracht gezogen würden. Wie bereits erwähnt, spielen hierbei zwei Variablen eine zentrale Rolle: *Alpha* beschreibt die untere Schranke (engl. lower bound) des maximierenden Spielers und entspricht dem Wert, den dieser Spieler momentan mindestens erreichen wird. Die Variable *Beta* hingegen enthält die obere Schranke (engl. upper bound) des minimierenden Spielers und gibt den Wert an, der höchstens erreicht wird (vom minimierenden Spieler). Aufgrund des rekursiven Charakters des Alpha-Beta Algorithmus' lassen sich die Werte für Alpha und Beta in jedem Knoten des Baumes ändern und mit den aktuellen Werten vergleichen, wodurch Cutoffs nicht nur auf den Wurzelknoten beschränkt sind, sondern an jeder Stelle des Spielbaum erfolgen können.

Aus Sicht des maximierenden Spielers werden folgende Schritte zur Abarbeitung eines Spielknotens unternommen: Zunächst ermittelt die Suche alle möglichen legalen Züge, die anschließend sukzessive ausprobiert werden. Liefert die Suche für einen der darauffolgenden Spielzustände einen Wert größer oder gleich Beta, wird der aktuelle Knoten sofort verlassen und Beta zurückgegeben. Es handelt sich hierbei um einen *Fail-High*. Der zurückgegebene Wert entspricht lediglich einer unteren Schranke des tatsächlichen *spieltheoretischen Wertes* der Position.

Ein *Fail-Low* tritt in einem Knoten dann auf, wenn Alpha kein einziges mal angepasst wurde und daher kein Zug das bisherige Ergebnis verbessern konnte. Beim Verlassen des Knotens gibt die Routine daher Alpha, die obere Schranke für den exakten Wert der Position, zurück. Einzig und allein Werte zwischen Alpha und Beta sind exakt, in allen anderen Fällen werden lediglich Schranken ermittelt, da die präzisen Werte für den Ausgang des Spiels völlig irrelevant sind.

Bruce Moreland beschreibt in seiner Einführung zur Schachprogrammierung den obigen Sachverhalt sehr treffend und anschaulich [7]:

> *A fail-high indicates that the search found something that was "too good". What this means is that the opponent has some way, already found by the search, of avoiding this position, so you have to assume that they'll do this. If they can avoid this position, there is no longer any need to search successors, since this position won't happen. A fail-low indicates that this position was not good enough for us. We will not reach this position, because we have some other means of reaching a position that is better. We will not make the move that allowed the opponent to put us in this position.*

Analog zum bereits Genannten lassen sich minimierende Knoten untersuchen. Details hierzu können dem vereinfachten Pseudo-Code in Listing 4.1 und 4.2 oder dem Quelltext (*c4.AlphaBetaAgent*) im Anhang entnommen werden. Einzelheiten zum Pseudocode werden im Laufe dieses Kapitels erläutert.

```
double max(double alpha, double beta) {
    if(hasWin(max)) return +1;
    if(isDraw())return 0;
    if(bookEntryAvailable()) return bookValue();
    if(TransTableLookup(board, mirroredBoard))
        return transpositionValue;
    generateMoves(max);
    sortMoves(max);
    if(isSymmetricPosition())
        removeSymMoves();
    while(movesLeft(max)) {
        makeMove(max);
        value = min(alpha, beta);
        takeMoveBack(max);
        if(value >= beta) {
            putEntryInTranspositionTable();
            return beta;
        }
        if(value > alpha)
            alpha = value;
    }
    putEntryInTranspositionTable();
    return alpha;
}
```

Listing 4.1: Pseudo-Code der Alpha-Beta-Suche für den maximierenden Spieler. Zugriffe auf die Eröffnungsbücher und Transpositions-Tabellen finden ausschließlich in Knoten des maximierenden (anziehenden) Spielers statt.

```
double min(double alpha, double beta) {
    if(hasWin(min)) return -1;
    if(enhancedTranspositionCutoff(board, mirroredBoard))
        return ETCValue;
    generateMoves(min);
    sortMoves(min);
    if(isSymmetricPosition())
        removeSymMoves();
    while(movesLeft(min)) {
        makeMove(min);
        value = max(alpha, beta);
        takeMoveBack(min);
        if(value <= alpha)
            return alpha;
        if(value < beta)
            beta = value;
    }
    return beta;
}
```

Listing 4.2: Pseudo-Code der Alpha-Beta-Suche für den minimierenden Spieler. Die Suche nach Enhanced Transposition Cutoffs (ETCs) findet ausschließlich in den Knoten des minimierenden (nachziehenden) Spielers statt.

4.2 Repräsentation des Spielfeldes

4.2.1 Einführung

Die *Spielfeldrepräsentation* ist ein elementarer Bestandteil des Agenten. Da sehr große Teile des Programmes diese Datenstruktur als Basis benötigen, ist besondere Sorgfalt bei der Spezifikation dieser geboten. Dies gilt insbesondere für Baumsuchverfahren wie der Alpha-Beta-Suche, bei denen sehr viele Operationen auf dieser Datenstruktur ausgeführt werden.

Die vermutlich einfachste Darstellung eines Spielfeldes wird in den meisten Spielen durch die Verwendung eines ein- oder zweidimensionalen Feldes (Array) erreicht. Für *Vier Gewinnt* könnte beispielsweise eine 7x6-Integer-Matrix Verwendung finden. Auch für weniger komplexe Spiele wie *Tic Tac Toe* ist dieser Ansatz durchaus zu empfehlen, da Effizienz keine besondere Rolle spielt und die Repräsentation des Spielfeldes die Programmierung in vielen Teilen vereinfacht.

Für komplexere Spiele wie *Vier Gewinnt*, *Mühle*, *Dame* oder *Schach* ist solch eine Darstellung oft unvorteilhaft. In diesen Spielen muss ein großer Wert auf die Effizienz der Algorithmen und der zugrundeliegenden Datenstrukturen gelegt werden. Reduziert man den Minimax- oder auch den Alpha-Beta-Algorithmus auf die grundlegendsten Operationen, ist schnell ersichtlich, dass vor allem Spielfeldmanipulationen (Züge ausführen / zurücknehmen) und die Identifizierung von terminalen Spielzuständen (Gewinne / Niederlagen oder Unentschieden) eine zentrale Rolle spielen. Diese Operationen finden allesamt auf der Datenstruktur des Spielfeldes statt.

Für das Spiel *Vier Gewinnt* kann insbesondere die Suche nach Siegen bzw. Niederlagen (geschlossene Viererketten), die in praktisch allen Knoten des Spielbaumes stattfindet, zum Flaschenhals des Agenten werden. Gerade diese Suche nach geschlossenen Viererketten ist bei der Verwendung von Feldern (Arrays) vergleichsweise aufwendig, vor allem wenn diese viele Millionen Mal hintereinander ausgeführt werden muss.

Aus diesem Grund wurden in dem Alpha-Beta-Agenten dieser Arbeit sogenannte *Bitboards* verwendet, die einige grundlegende Vorteile im Vergleich zu den einfachen Feldern besitzen. Vor allem die zeitkritische Suche nach Viererketten kann mithilfe dieser Repräsentation effizient durchgeführt werden.

4.2.2 Bitboards

In der Schachprogrammierung werden *Bitboards* (häufig im Zusammenspiel mit einfachen Feldern) schon länger zur Spielfeldrepräsentation eingesetzt. Insbesondere die Anzahl von 64 Feldern des Schachbretts macht die Verwendung von Bitboards interessant, da zahlreiche Systeme mittlerweile über 64-Bit-Architekturen verfügen und 64-Bit-Variablen direkt mit einem CPU-Befehl verarbeitet werden können. Auch Spiele wie *Othello* oder *Dame* (Checkers) mit 64 Feldern könnten hiervon profitieren.

Im Wesentlichen sind Bitboards einfache Variablen, deren Betrachtung jedoch auf der Bit-Ebene erfolgt. In der Regel entsprechen die individuellen Bits einer Variable einer Position auf dem Spielfeld. Durch das Setzen oder Löschen der Bits können einzelne Spielsteine an den entsprechenden Position gesetzt oder entfernt werden. Um etwas Zeit während der Baumsuche zu sparen, ist hierfür die Verwendung von bereits vorgefertigten Masken sinnvoll. Die Laufzeitvorteile, die sich hierdurch ergeben, sind nicht zu unterschätzen.

Da die Bitboards keinerlei Information zur Art des Spielsteins enthalten, können in einem Bitboard nur Spielsteine gleichen Typs gehalten werden. Beispielsweise könnten beim *Schachspiel* alle Bauern des weißen Spielers in einer 64-Bit Variable gehalten werden, jedoch keine Läufer, die ein eigenes Bitboard benötigten.

Vier Gewinnt kennt nur zwei Typen von Spielsteinen, wodurch zwei Bitboards (eins für jeden Spieler) nötig werden. Pro Bitboard sind wiederum 42 Bits erforderlich. Zweckmäßig ist daher die Verwendung von 64-Bit-Variablen (z.B. vom Typ Double oder Long in Java), von denen beispielsweise die niederwertigsten 42 Bit genutzt werden (wie in diesem Projekt geschehen). Zwei 64-Bit Variablen reichen daher aus, um ein *Vier Gewinnt*-Spielfeld vollständig zu beschreiben. Das Setzen eines Spielsteines kann mithilfe einer bitweisen ODER-Operation erfolgen, das Löschen mit einer bitweisen UND-Operation[11].

Hendrik Baier beschreibt in seiner Bachelor-Thesis wie die Gewinnsuche erfolgen kann [8]: Zur Erkennung einer Gewinnstellung ist ausschließlich der aktuelle Halbzug relevant. Aufgrund dessen ist nur eine lokale Suche nach Viererketten im Umfeld der belegten Zelle notwendig. Anhand einiger bitweiser UND-Operationen, die auf Bitboard des betreffenden Spielers und den entsprechenden Bitmasken (eine für jede mögliche Viererkette) ausgeführt werden, sind etwaige Viererketten schnell zu ermitteln. In dieser Arbeit ist die Gewinnsuche ähnlich umgesetzt, mit der Ausnahme, dass

[11] Da Spielfeldmanipulationen während der Suche sehr oft ausgeführt werden, ist es vorteilhaft, wenn die entsprechenden Masken für alle 42 Spielfelder bereits vorgefertigt sind. Dies kann einiges an Rechenzeit einsparen.

hier mit den invertierten Bitboards gearbeitet wird[12], auch die Darstellung der Bitboards ist etwas anders gewählt worden.

Die Anzahl der UND-Operationen hängt allerdings von der Platzierung des Spielsteins ab. Generell lässt sich sagen, dass man für zentrale Felder mehr Operationen benötigt als für äußere, da in den zentralen Feldern grundsätzlich mehr potentielle Viererketten möglich sind.

4.2.3 Vor- und Nachteile von Bitboards

Insgesamt können eine Reihe von Vorteilen von Bitboards gegenüber Arrays ausgemacht werden (bezogen auf das Spiel *Vier Gewinnt*; nicht alle Vorteile sind zwangsläufig auf andere Spiele übertragbar):

- Viele Operationen können effizienter ausgeführt werden, da diese mehrere Felder des Spielbretts gleichzeitig miteinbeziehen können. Beispielhaft genannt seien hier die Erkennung von Gewinnstellungen und Remis, das Spiegeln von Spielfeldern an der mittleren Spalte und die Bestimmung von Drohungen.
- Das Kopieren des Spielfeldes kann deutlich schneller erfolgen, da lediglich zwei 64-Bit Variablen das Spielfeld repräsentieren.
- Geringerer Speicherbedarf in Zusammenhang mit Transpositionstabellen.

Allerdings kann die Verwendung von Bitboards einige entscheidende Nachteile nach sich ziehen, die nicht verschwiegen werden sollten:

- Hoher Programmieraufwand: Teilweise wurden auch Teile des Quelltextes mit weiteren Programmen erzeugt.
- Schlechte Lesbarkeit des Quelltextes.
- Wartung des Programmes ist nur schwer möglich.
- Fehleranfälligkeit und mühsame Fehlersuche.
- Sehr langer Programmcode: Alles in allem sind nahezu 7000 Zeilen Quelltext zur Realisierung dieses Agenten notwendig gewesen.

[12] Die Invertierung des Bitboards erspart eine anschließende Vergleichsoperation pro Maske. Stattdessen kann direkt das Zero-Flag im Statusregister der CPU für den bedingten Sprung genutzt werden; Details hierzu können dem Quelltext entnommen werden.

4.3 Zugsortierung

4.3.1 Allgemein

Um eine möglichst hohe Effizienz des Alpha-Beta-Algorithmus' zu erreichen, ist vor allem eine gute *Zugsortierung* nötig. Im Gegensatz zum Minimax-Algorithmus, der alle Knoten des Suchbaums untersuchen muss, können beim Alpha-Beta-Algorithmus große Teile des Baums beschnitten werden, wenn frühzeitig die besten Züge ausprobiert werden. Der Grund hierfür ist, dass die Kontrahenten jeweils versuchen ihr eigenes Ergebnis zu maximieren und das des Gegners zu minimieren. Je mehr sich die aktuellen Resultate den spieltheoretischen Werten annähern, desto mehr Teilbäume können abgeschnitten werden, da diese keinen Beitrag zur Bestimmung der exakten Werte mehr leisten.

Cutoffs in höheren Ebenen des Suchbaums schneiden hierbei größere Teilbäume ab. Da sich der Rechenaufwand aufgrund der geringeren Zahl an Knoten in diesen Ebenen in Grenzen hält, sind dort ausgefeilte Algorithmen für die Zugsortierung denkbar, die zwar die Rechenzeit pro Knoten erhöhen, aber im Gegenzug höhere Cutoff-Raten erreichen. Wird dies in der Implementierung vernünftig umgesetzt, sind deutliche Laufzeitverbesserungen möglich. Daher sollte man, bevor am Programm technische Optimierungen und Ähnliches vorgenommen werden, zunächst versuchen, die Zugsortierung weiter zu verbessern. Dies bietet deutlich mehr Potential.

Die Alpha-Beta-Suche dieses Agenten ist in zwei Stufen unterteilt. In der ersten Stufe ist die Realisierung von laufzeitintensiven Prozeduren möglich, da die Anzahl der durchsuchten Knoten vergleichsweise überschaubar ist. Die zweite Stufe dagegen ist sehr statisch gehalten um die Rechenzeit pro Knoten zu minimieren. Im Folgenden werden die Techniken zur Zugsortierung in beiden Stufen etwas genauer diskutiert.

4.3.2 Einfache Zugsortierung

Generell lässt sich sagen, dass Halbzüge in die mittleren Spalten des Spielfeldes im Regelfall zu bevorzugen sind. Das Setzen in zentrale Spalten erhöht die Wahrscheinlichkeit, im weiteren Verlauf des Spiels Drohungen zu erzeugen, die für den Spielausgang entscheidend sind.

In der zweiten Stufe der Suche findet diese simple Sortierung nach wie vor Verwendung, da sie vollkommen ohne dynamische Elemente auskommt und dadurch sehr effizient zu implementieren ist. Bisher waren keine weiteren Maßnahmen erfolgreich, um die Zugsortierung der zweiten Stufe zu verbessern. Dies liegt vor allem daran, dass Cutoffs in tieferen Ebenen des Suchbaumes nicht ausreichend Knoten einsparen, um den nötigen Zusatzaufwand zu kompensieren.

Für die erste Stufe sind allerdings noch weitere Verfeinerungen des oben genannten Ansatzes möglich, um zusätzliche Laufzeitreduzierungen zu erreichen. Wie in [8] richtig

herausgestellt wird, können zentrale Felder an mehr Viererketten beteiligt sein, als am Rand Liegende. So kann etwa neben der Spalte auch die Zeile einer Spielfeldzelle zur Einschätzung der erwarteten Qualität eines Halbzuges herangezogenen werden. Beispielsweise ist ein Halbzug in die Zelle c3 in der Regel einem in die Zelle d6 überlegen, da sich durch die erstgenannte theoretisch eine größere Zahl Viererketten legen lassen.

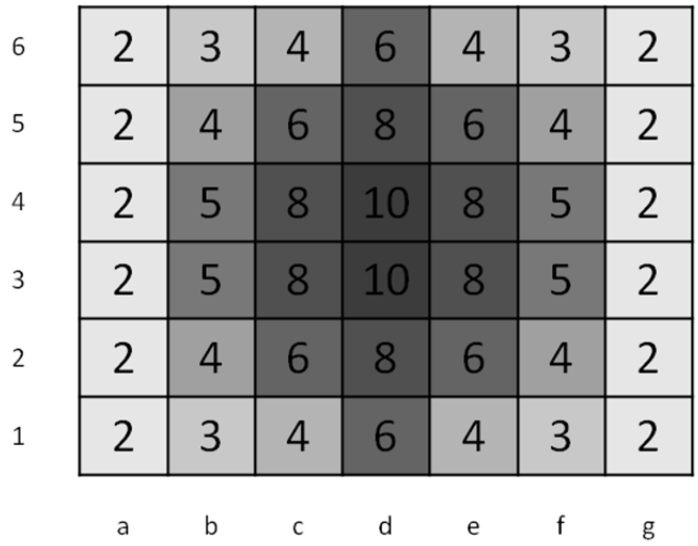

Abbildung 4.1: Anzahl der Viererketten, an denen die jeweiligen Spielfeldzellen beteiligt sein können (mit Ausnahme der vertikalen Ketten)[13]

Als weiterer Aspekt zur Einschätzung der Qualität eines Halbzuges können die benachbarten Felder einer Zelle dienen. Zellen mit vielen benachbarten gegnerischen Steinen sind für einen Spieler oft wertlos, auch wenn diese zentral gelegen sind. Halbzüge in solche Felder sollten daher nach Möglichkeit von beiden Spielern vermieden werden. *Baier* [8] versuchte diese Idee mithilfe von zwei Arrays (je ein Array pro Spieler) umzusetzen, indem die Bewertungen der einzelnen Zellen während der Suche dynamisch angepasst wurden. Aus Effizienzgründen verwarf er diesen Ansatz jedoch wieder.

An dieser Stelle wird eine etwas andere Herangehensweise diskutiert, die sehr gute Ergebnisse lieferte: Hierzu gleicht man alle möglichen Viererketten, an denen die betreffende Zelle beteiligt ist, mit den Spielsteinen des Gegners ab und wertet den entsprechenden Zug bei jeder Übereinstimmung (die Steine des Gegners machen eine vollständige Belegung dieser Viererkette durch den aktuellen Spieler unmöglich) ab[14]. Dies wird in jedem Knoten wieder neu durchgeführt, kann aber mithilfe von bitweisen

[13] Ein kleines Programm zur Bestimmung der Werte befindet sich im Anhang.

[14] Wie sich herausstellte, lieferte die Zugsortierung etwas bessere Ergebnisse, wenn die vertikalen Vierer-Ketten außer Acht gelassen wurden.

UND-Operationen und den entsprechenden Masken besonders effizient erfolgen. Weiterhin findet diese Art der Zugsortierung lediglich in wurzelnahen Knoten Anwendung.

4.3.3 Erzeugung von Drohungen

Der vermutlich wichtigste Optimierungsansatz ist allerdings bis jetzt ausgelassen worden: Vor allem in der frühen und mittleren Phase eines Spiels ist es für die Spieler wichtig Drohungen aufzubauen. Gelingt dies einem der beiden Spieler, kann dieser den anderen unter Umständen im weiteren Verlauf des Spiels unter *Zugzwang* setzen und das Spiel gewinnen. Jedoch sind nicht immer alle Drohungen eines Spielers hilfreich, um den Gegner letztendlich zu schlagen. Beide Spieler müssen unterschiedliche Strategien zur Erzeugung von Drohungen verfolgen. Vor allem die Position (i.d.R. ist die Reihe ausschlaggebend) und die Anordnung einer oder mehrerer Drohungen entscheiden darüber, wie eine Stellung zu beurteilen ist.

Victor Allis ist in seiner Masterarbeit [1] ausführlich auf diese Thematik eingegangen. Jedoch sind die aufgestellten Regeln so komplex und abstrakt, dass sie sich nur sehr schwer auf die Zugsortierung übertragen lassen.

Da die Zugsortierung ohnehin nicht in allen Fällen die optimale Zugreihenfolge für einen Knoten liefern kann, sind an dieser Stelle auch einige Verallgemeinerungen ausreichend, die eine möglichst große Zahl an Situationen abdecken:

1. Grundsätzlich gewinnt der Spieler, der *mehr* Drohungen in ungeraden Reihen (ungerade Drohungen) besitzt.
2. Dem anziehenden Spieler sind im Regelfall nur *ungerade Drohungen* von Nutzen.
3. *Gerade Drohungen* nützen ausschließlich dem nachziehenden Spieler. Sobald der anziehende Spieler mindestens eine ungerade Drohung besitzt, sind diese jedoch *unbrauchbar*.

Unter Beachtung dieser allgemeinen Aussagen, ist es möglich die Zugsortierung folgendermaßen anzupassen: Für den anziehenden Spieler werden Züge, die eine ungerade Drohung erzeugen, bevorzugt untersucht, unabhängig von der Position des gesetzten Steines. Dies gilt auch für den Nachziehenden, mit der Ausnahme, dass zusätzlich auch gerade Drohungen berücksichtigt werden. Ist es im aktuellen Knoten möglich mehrere Drohungen zu erzeugen, sind Züge in die mittleren Spalten vorrangig zu behandeln.

In vielen Fällen ist es so, dass einer der beiden Spieler unter *direkten Zugzwang* (Abwenden einer sofortigen Niederlage) gesetzt werden muss, damit der Gegner eine geeignete Drohung erzwingen kann. Auch solche Züge werden von der Zugsortierung erkannt und vor den verbleibenden Zügen ausprobiert.

Da die Suche nach Drohungen vergleichsweise aufwendig ist, kann diese nur in höheren Ebenen der Alpha-Beta-Suche erfolgen. Dennoch sind die Laufzeitvorteile be-

achtlich, die durchschnittliche Suchzeit des Agenten halbiert sich aufgrund dieser Maßnahme[15].

4.3.4 Vorsortierung mithilfe eines N-Tupel-Systems

Werden N-Tupel-Systeme zur Approximierung der Spielfunktion eingesetzt, sind in aller Regel keine exakten Stellungsbewertungen möglich. Ein gutes System kann aber in vielen Fällen zumindest eine grobe Einschätzung liefern, sodass die Qualität der möglichen Folgezustände richtig eingeordnet wird. Dieser Sachverhalt kann zur Optimierung der Zugsortierung ausgenutzt werden: Die Ausführungsreihenfolge der Züge in einem Knoten ist für die Korrektheit der Alpha-Beta Suche irrelevant, eine schlechte Zugsortierung wirkt sich lediglich auf die Laufzeit der Suche negativ aus. In der Hoffnung, dass das eingesetzte N-Tupel-System die Folgezustände eines Knoten bzgl. ihrer Qualität in vielen Fällen richtig einschätzen kann, ist ein Einsatz zur Vorsortierung der Züge eine gute Alternative zu dem in Abschnitt 4.3.2 beschrieben Verfahren.

Testweise wurde eine N-Tupel-System mit 70 N-Tupeln (jeweils sieben Abtastpunkte) trainiert, das die einfache Zugsortierung in wurzelnahen Knoten ersetzte. Die Rechenzeit zur Analyse des leeren Spielfeldes konnte mithilfe dieser Maßnahme auf ein Drittel der ursprünglichen Zeit verkürzt werden und auch für viele andere Stellungen konnte man zum Teil deutliche Verbesserungen beobachten. Da sich in gewissen Fällen auch die Suchzeit erhöhte, ist jedoch noch eine umfassende Untersuchung der unterschiedlichen Ergebnisse nötig, die aus Zeitgründen nicht mehr vorgenommen wurde.

4.3.5 Zusammenfassung

Je nach Suchtiefe, kann die Zugsortierung mehr oder weniger laufzeitintensiv konstruiert werden. Mithilfe einer *dynamischen Zugsortierung* sind hohe Cutoff-Raten möglich. Aufgrund des erhöhten Aufwandes für die Sortierung ist der Einsatz allerdings nur in höheren Ebenen des Suchbaumes denkbar. Die *statische Zugsortierung* liefert in vielen Fällen nicht so gute Resultate wie die dynamische, minimiert aber die Rechenzeit pro Knoten. Daher ist eine statische Zugsortierung vor allem für tiefer liegende Suchbaum-Ebenen geeignet. Durch die Zwei-Stufen-Struktur der Alpha-Beta Suche können beide Verfahren zum Einsatz kommen. In der ersten Stufe findet die dynamische Zugsortierung Anwendung, die die Züge nach folgender Prioritätenliste sortiert[16]:

[15] Aus Effizienzgründen und zur Vermeidung von Fehlern wurde der zugehörige Quelltext maschinell erzeugt, was einen sehr langen Code zur Folge hat (über 2000 Zeilen). Das generierende Programm ist im Anhang aufgeführt.

[16] Halbzüge, die das Spiel sofort beenden oder mindestens zwei unmittelbare Drohungen erzeugen (vgl. Zwickmühle) haben selbstverständlich die höchste Priorität. Da diese in der Implementierung nicht mehr ausgeführt werden (das Endresultat ist ja bereits bekannt), berücksichtigt die Zugsortierung solche Züge nicht weiter.

Bevor die eigentliche Zugsortierung einsetzt, werden zunächst überflüssige Züge aussortiert. Ist eine Stellung symmetrisch, liefert die Spiegelung an der mittleren Achse daher dieselbe Position, ist es ausreichend, lediglich die ersten vier Spalten zu untersuchen[17].

1. Züge, die eine (nicht unmittelbare) Drohung erzeugen, haben die höchste Priorität. Für den anziehenden Spieler sind nur ungerade Drohungen relevant (Abschnitt 4.3.3).
2. Auch Züge, die den Gegner unter direkten Zugzwang setzen und im Anschluss die Erzeugung einer eigenen geeignete Drohung erlauben, haben eine hohe Priorität (Abschnitt 4.3.3).
3. Anschließend werden die verbleibenden Züge untersucht. Zentral liegende Zellen erhalten, unter Berücksichtigung der benachbarten gegnerischen Steine, eine höhere Priorität als am Rand liegende (Abschnitt 4.3.2).
4. Hierbei sortiert die Suche Züge nach hinten, die unter eigene Drohungen ziehen. Dies geschieht angesichts der Tatsache, dass der Gegner diese Drohung im nächsten Zug neutralisiert und infolgedessen nutzlos macht. Auch hier sind nur ungerade Drohungen für den anziehenden Spieler von Bedeutung.

Die zweite Stufe der Alpha-Beta Suche verwendet die triviale, weitestgehend statische Zugsortierung. Von der mittleren Spalte ausgehend, werden die Züge in Richtung der äußeren Spalten ausgeführt. Lediglich Züge unter eigene Drohungen werden vermieden und erst zum Ende hin untersucht.

[17] Diese Maßnahme ist insbesondere für Spielstellungen mit wenigen Spielsteinen sehr effektiv.

4.4 Transpositionstabellen

Die Verwendung von *Transpositionstabellen* (Hashtabellen) ist eine weitere Möglichkeit, um eine Beschleunigung der Baumsuche zu erreichen. Sie werden hauptsächlich dazu eingesetzt, um mehrfache Analysen (Untersuchung ganzer Teilbäume) einzelner Positionen während der Suche zu vermeiden. Im Zusammenhang mit *Iterative Deepening* ist auch die Optimierung der Zugsortierung mithilfe von Transpositionstabellen denkbar. In diesem Projekt wird Iterative Deepening allerdings nicht für die Alpha-Beta Suche eingesetzt, da damit keine Laufzeitvorteile erreicht werden konnten. Unabhängig vom betrachteten Brettspiel, erreicht man im Allgemeinen durch den Einsatz von Transpositionstabellen in Verbindung mit der Baumsuche eine enorme Einsparung an Knoten, sodass eine Halbierung der Suchzeit durchaus realistisch ist.

4.4.1 Permutationen einzelner Zug-Sequenzen

In vielen Spielen führen Änderungen an der Reihenfolge der Halbzüge innerhalb einer Zugfolge oft zu ein und derselben Stellung. Die unterschiedlichen Anordnungen einer Zugfolge können auch als *Permutationen* bezeichnet werden, wobei eine Permutation eine *bijektive Abbildung* einer Menge von Halbzügen auf sich selbst darstellt[18]. Zu beachten ist, dass nicht alle Permutationen einer Zugfolge zwangsläufig zu derselben Spielstellung führen. Einzelne Permutationen können unmögliche Halbzüge enthalten oder aufgrund der Spielcharakteristika in anderen Spielzuständen enden. Beim *Schachspiel* ist es z.B. nicht möglich eine Figur auf ein Feld zu ziehen, dass bereits durch eine Figur des gleichen Spielers belegt ist; dies könnte bei gewissen *Zugumstellungen* passieren.

Weiterhin resultiert aus einer einzelnen Transposition, bei der ein Halbzug des einen Spielers mit einem Halbzug des Gegners getauscht wird, eine illegale Zugfolge, da zwei Halbzüge des gleichen Spielers direkt aufeinanderfolgen würden. Besser ist es daher, die Halbzüge der einzelnen Spieler in getrennten Zuglisten zu führen. Aber auch nicht alle Permutationen der getrennten Zug-Sequenzen führen notwendigerweise immer zu identischen Spielzuständen. Beim Spiel *Vier Gewinnt* wird dies etwa durch die Schwerkrafts-Regel des Spiels verhindert, bei *Schach* oder *Dame* z.B. durch die schlagenden Züge. *Tic Tac Toe* und *Gomoku* hingegen stellen Spiele dar, für die ausnahmslos alle Permutationen einer Zugfolge (jeweils für beide Spieler getrennt behandelt) zur selben Spielstellung führen.

[18] Die einzelnen Permutationen können mithilfe einer endlichen Zahl von Transpositionen erzeugt werden, wobei eine Transposition als Permutation, die genau zwei Elemente (Halbzüge) miteinander vertauscht, definiert ist. Der Begriff der Transpositionstabelle rührt allerdings aus dem Englischen. Zugumstellungen beim Schachspiel werden dort oft als *transpositions* (engl.) bezeichnet.

4.4.2 Ein Hash-Verfahren nach Albert L. Zobrist

Da unter Umständen mehrere tausend oder sogar millionen Knoten des Spielbaums pro Sekunde untersucht werden, ist beim Einsatz von Transpositionstabellen eine besondere Effizienz des *Hash-Verfahrens* erforderlich, es wird daher eine Funktion gesucht, die einen Spielzustand ohne besonderen Aufwand auf einen *Hashwert* (Index der Tabelle) abbildet. Solch ein Verfahren hat *Albert L. Zobrist* [9] bereits im Jahre 1970 vorgestellt. Es eignet sich insbesondere für Brettspiele und ist mittlerweile in der Schachprogrammierung sehr weit verbreitet. Im Folgenden wird näher darauf eingegangen.

Die zentrale Operation für das *Zobrist-Hashing* ist das *bitweise Exklusiv-ODER* (XOR), mit dem n-stellige Dualzahlen verknüpft werden, um einen *Schlüssel* für eine Stellung zu erhalten. Im Allgemeinen legt man hierzu eine Liste von Zufallszahlen an, von der jede einem Spielstein an einer bestimmten Position entspricht. Je Spieler muss ein eigener Satz von Zufallszahlen erzeugt werden. Allgemein kann man die Anzahl nötiger Zufallszahlen in etwa folgendermaßen beschreiben:

$$N_{Zufall} = Anzahl\ Spieler \times \frac{Anzahl\ Figurtypen}{Spieler} \times Anzahl\ Felder \qquad (4.1)$$

wobei gewisse Sonderzüge evtl. gesondert codiert werden müssen.

Für *Vier Gewinnt* wären demnach 2 x 42 Zufallszahlen notwendig. Beim Hinzufügen eines Spielsteines aktualisiert man den aktuellen Schlüssel ganz einfach, indem dieser mit der entsprechenden Zufallszahl XOR-verknüpft wird. Aber auch das Entfernen oder Verschieben von Spielsteinen kann problemlos realisiert werden. Dies und weitere Besonderheiten sind bei Betrachtung einiger grundlegender Eigenschaften der XOR-Verknüpfung (Notation: $x \nleftrightarrow y$) schnell ersichtlich:

1. $(x \nleftrightarrow y) \nleftrightarrow z = x \nleftrightarrow (y \nleftrightarrow z)$ (*Assoziativität*) und $x \nleftrightarrow y = y \nleftrightarrow x$ (*Kommutativität*): Für die Erzeugung des Schlüssels ist die Reihenfolge der einzelnen Halbzüge nicht relevant. Permutationen einer Zugfolge führen zu dem gleichen Schlüssel (sofern auch die Stellungen identisch sind).
2. $x \nleftrightarrow x = 0$: Dies erlaubt die Entfernung eines Spielsteins aus dem Zobrist-Schlüssel[19].
3. Die bitweise XOR-Verknüpfung von beliebig vielen n-stelligen zufälligen Dualzahlen liefert als Ergebnis auch wiederum eine n-stellige Zufallszahl[20].

Außerdem erlaubt das Zobrist-Hashing die inkrementelle Anpassung des Schlüssels. Dies ist vor allem während der Baumsuche von großem Vorteil, da in jedem Knoten nur

[19] Um einen Spielstein zu verschieben wird dieser zunächst aus dem Schlüssel entfernt und anschließend an einer anderen Stelle wieder eingefügt. Dafür sind zwei bitweise XOR-Operationen notwendig.

[20] Dies ist längst nicht bei allen Operationen der Fall (beispielsweise für die bitweise UND-Verknüpfung).

sehr wenige XOR-Operationen nötig sind, um den Schlüssel zu aktualisieren. Für *Vier Gewinnt* Suche ist sogar nur eine einzige XOR-Operation je Knoten notwendig.

Mithilfe einer einfachen *Hashfunktion*, z.B. einer Modulo-Funktion, bestimmt man anhand des Schlüssels den *Index* innerhalb der Transpositionstabelle. In gewissen Fällen kann die Modulo-Funktion auch in eine bitweise UND-Operation überführt werden. Dies ist immer der Fall, wenn man Transpositionstabellen der Größe 2^k einsetzt.

Eindeutige Schlüssel lassen sich mit diesem Verfahren allerdings nicht erzeugen; die Abbildung der Spielzustände auf die Menge der Schlüssel ist *nicht injektiv*. So traten beim Spiel *Vier Gewinnt* vereinzelt Fehler auf, wenn 32-Bit Schlüssel verwendet wurden. Bei der Verwendung von 64-Bit Schlüsseln konnten keine Fehler mehr beobachtet werden, auszuschließen sind sie dennoch nicht; dennoch wird die Suche durch deren Einsatz deutlich verlangsamt.

4.4.3 Konzeption der Transpositionstabellen

Wie bereits erwähnt, ist die Alpha-Beta Suche für *Vier Gewinnt* aus Performancegründen in zwei Stufen unterteilt. Auch der Einsatz einer zweistufigen Transpositionstabelle brachte deutliche Laufzeitvorteile. Dies liegt vor allem daran, dass Einträge für Knoten höherer *Ebenen* nicht durch die, der tiefer liegenden, überschrieben werden. Da vor allem Treffer in der Transpositionstabelle der ersten Stufe für eine beachtliche Knoteneinsparung sorgen, macht die Trennung in zwei Stufen daher durchaus Sinn[21].

Für jeden Knoten wird zunächst geprüft, ob ein Eintrag in der entsprechenden Transpositionstabelle vorhanden ist. Sollte dies der Fall sein, kann die Suche in vielen Fällen an dieser Stelle unterbrochen und der entsprechende Wert zurückgegeben werden. Andernfalls wird die Suche unverändert fortgesetzt und das Resultat für den Knoten anschließend in die zugehörige Transpositionstabelle eingetragen. Da die *Hashfunktion* nicht injektiv ist, bekommen unter Umständen mehrere Schlüssel denselben Index in der Transpositionstabelle zugeordnet. Solche *Kollisionen* können nicht vermieden werden, es gibt jedoch einige Lösungsstrategien um diese zu behandeln. Allerdings wurde für dieses Projekt nur das *Lineare Sondieren* getestet. Eine Reduzierung der Laufzeit war hiermit nicht möglich, sodass in der aktuellen Version des Programmes bereits vorhandene Einträge im Falle einer Kollision einfach überschrieben werden.

Jeder Eintrag einer Transpositionstabelle besteht aus drei Elementen:

1. Stellungs-Schlüssel: Da auch während der Lesevorgänge Kollisionen auftreten können, ist es notwendig die zugehörige Position mit abzuspeichern.

[21] Wird die aktuelle Suchtiefe in jeden Eintrag notiert, ist eine noch genauere Abstufung möglich. Jedoch kann es passieren, dass Einträge nicht mehr überschrieben werden, obwohl sie für die Suche mittlerweile irrelevant sind.

Hier bietet es sich an, den Zobrist-Schlüssel als Identifizierung der Stellung zu verwenden und für jeden Eintrag in der Transpositionstabelle abzulegen. Wie weiter oben bereits erwähnt, kann es dennoch dazu kommen, dass aus zwei unterschiedlichen Stellungen identische Zobrist-Schlüssel erzeugt werden. Dies versucht man zu vermeiden, indem die Schlüssellänge geeignet gewählt wird. Vollständig ausschließen lassen sich Fehler jedoch nur, wenn die komplette Stellung, also die Bit-Boards beider Spieler, für die jeweiligen Einträge in der Transpositionstabelle gespeichert sind[22].

2. Wert der Stellung.
3. Typ des Knotens: Da der Wert eines Knoten nur dann exakt ist, wenn dieser zwischen Alpha und Beta liegt (siehe Abschnitt 4.1), muss zusätzlich zum Wert des Knoten die Information des Knoten-Typs in jedem Eintrag der Transpositionstabelle enthalten sein. Bei dem Wert kann es sich wie bereits erwähnt um ein exaktes Resultat, einer oberen oder einer unteren Schranke handeln.

4.4.4 Enhanced Transposition Cutoffs

Im Allgemeinen wird, bevor die eigentliche Alpha-Beta Suche einsetzt, in jedem Knoten zunächst die Transpositionstabelle nach einem passenden Eintrag untersucht. In vielen Fällen ist kein solcher Eintrag vorhanden, sodass eine ausführliche Analyse der Stellung nötig wird. Diese versucht man mit den *Enhanced Transposition Cutoffs* (ETC) [10] zu vermeiden. Der Grundgedanke der dahinter steht ist folgender: Wenn die Transpositionstabelle keinen geeigneten Eintrag zur aktuellen Stellung enthält, ist es dennoch möglich, dass einzelne Folgestellungen darin gefunden werden. In der Hoffnung einen Cutoff zu erreichen, erzeugt man also die Schlüssel aller Folgestellungen und schlägt diese in der Tabelle nach. Häufig kann so ein Cutoff erreicht werden, ohne dass man den aktuelle Knoten weiter expandieren muss. Da Hash-Zugriffe aber häufig sehr zeitaufwändig sind, bringt dieses Verfahren oft nur in wurzelnahen Knoten einen wirklichen Vorteil. Im *Vier Gewinnt* Programm kommen ETCs daher nur in der ersten Stufe der Alpha-Beta Suche zum Einsatz. Durch deren Einsatz konnte die Suche um ca. 15 % beschleunigt werden.

4.4.5 Weitere Optimierungen

Wie zuvor bereits erwähnt wurde, kann die Ausnutzung von *Spiegel- oder Rotationssymmetrien* die Suche erheblich beschleunigen. Diesen Sachverhalt kann man auch für Zugriffe auf die Transpositionstabellen nutzen. Liefert eine Abfrage kein Ergebnis, ist oft eine Wiederholung mit gespiegelten oder rotierten Positionen erfolgreich. Im Allgemeinen ist das Erzeugen von symmetrischen Stellungen vergleichsweise aufwän-

[22] Für die Berechnung von Eröffnungsdatenbanken sind diese Fehler in jedem Fall zu vermeiden, um Inkonsistenzen innerhalb dieser zu verhindern.

dig, sodass dies nur bis zu einer gewissen Suchtiefe erfolgen kann. Beim *Vier Gewinnt* Spiel konnte mithilfe dieser Maßnahme die Suche vor allem für das leere Spielfeld und für Positionen mit wenigen Spielsteinen beschleunigt werden (ohne Einsatz der Eröffnungsdatenbanken).

Bei einem erhöhtem *Füllgrad* der Transpositionstabellen steigt die Anzahl der Kollisionen schnell an. Um dies zu vermeiden, sollte man versuchen, unnötige Eintragungen in die Tabellen gar nicht erst vorzunehmen. Wie sich herausgestellt hat, ist es nicht erforderlich, in jedem Knoten des *Vier Gewinnt* Suchbaums auf die Transpositionstabellen zuzugreifen. So konnte der Füllgrad der beiden Transpositionstabellen deutlich verkleinert werden, indem die Suche nur noch auf jeder zweiten Ebene des Suchbaumes die Tabellen zurate zieht. Dadurch wird die Anzahl an Eintragungen während der Suche nahezu halbiert und es treten deutlich weniger Kollisionen auf.

4.5 Eröffnungsdatenbanken

4.5.1 Einführung

Wie bei vielen anderen Spielen auch, ist ebenfalls für *Vier Gewinnt* der weitere Verlauf und der Ausgang eines Spiels bereits von der Eröffnungsphase abhängig. So muss beispielsweise der anziehende Spieler seinen ersten Stein in die mittlere Spalte setzen um das Spiel zu gewinnen. In allen anderen Fällen wird nur noch ein Unentschieden erreicht oder das Spiel sogar verloren. Aus diesem Grund muss ein starker Spieler die Eröffnung beherrschen um eigene Fehler zu vermeiden und Fehler des Gegners sofort bestrafen zu können. Oftmals sind aber insbesondere Spielstellungen der Eröffnungsphase nur sehr schwer zu analysieren, da eine hohe Suchtiefe notwendig ist, um die exakten *spieltheoretischen Werte* zu ermitteln.

Um die Eröffnungsphase zu meistern werden daher für viele Spiele *Eröffnungsbücher* genutzt, die häufig mit hohem Rechenaufwand erzeugt wurden. Diese können dem Computer-Programm i.d.R. einen spielerischen Vorteil verschaffen und sparen viel Rechenzeit während der Eröffnung. Für Spiele wie *Vier Gewinnt* ist ein weiterer Vorteil auszumachen: Der Zustandsraum nimmt im Laufe des Spieles immer weiter ab, sodass nach Verlassen des Eröffnungsbuches einzelne Stellungen oft ohne weitere Hilfe und in akzeptabler Zeit korrekt bewertet werden können.

Für *Vier Gewinnt* existieren einige solcher Datenbanken, die aber i.d.R. vom Programmierer für das eigene Programm erzeugt wurden und für andere daher nicht nutzbar sind.

4.5.2 8-Ply Eröffnungsdatenbank

Die erste umfangreiche Datenbank, die auch für andere Programmierer nutzbar ist, wurde von *John Tromp* im Jahre 1994 veröffentlicht [2]. Die Berechnung der Datenbank wurde auf mehreren Rechnern durchgeführt und dauerte 40.000 Stunden (ca. viereinhalb Jahre). Die meisten starken *Vier Gewinnt*-Programme – unter anderem *Mustrum*[23] und auch ein Programm von *John Tromp* selbst – verwenden diese Datenbank in irgendeiner Form.

Die Datenbank enthält alle *non-terminalen* Stellungen mit acht Steinen, jedoch sind alle Stellungen, die nach Spiegelung an der mittleren Spalte identisch sind, nur in einer Variante in der Datenbank vertreten. Von den insgesamt 67.557 Spielstellungen führen 44.473 (65.83%) Stellungen zum Sieg des Anziehenden, 16.635 (24.62%) zu einer Niederlage und 6.449 (9.55%) zu einem Unentschieden.

Alle Positionen, mit mindestens einer unmittelbaren Drohungen für einen der beiden Spieler, sind jedoch *nicht* berechnet worden. Dadurch ist die Datenbank nicht ganz vollständig.

[23] Kann von www.mustrum.de heruntergeladen werden.

Stellungen mit einer unmittelbaren Drohung für den Anziehenden müssen nicht in die Datenbank aufgenommen werden, da dieser ohnehin im nächsten Halbzug das Spiel gewinnt. Da aber Stellungen mit einer unmittelbaren Drohung für den Nachziehenden im nächsten Halbzug neutralisiert werden können und das Spiel danach regulär fortgesetzt wird, ist eine direkte Bewertung dieser Spielzustände mithilfe der Eröffnungsdatenbank nicht möglich.

Die noch fehlenden Positionen mussten daher noch nachträglich berechnet und der Datenbank hinzugefügt werden. Insgesamt fehlten 19.336 Stellungen, die mit einer frühen Version des Alpha-Beta-Agenten in ca. zwei Wochen auf einem Pentium4-Rechner ausgewertet wurden. Die jetzt vollständige 8-Ply Datenbank besteht aus 86.893 Spielstellungen und ordnet jeder Stellung den theoretischen Ausgang des Spiels (Sieg / Niederlage / Unentschieden aus Sicht des Anziehenden) zu.

So können alle Stellungen mit acht Steinen und weniger leicht bewertet werden, indem der Alpha-Beta-Algorithmus die Suche bei acht Steinen abbricht und das Ergebnis aus der Datenbank entnimmt. Dies ermöglicht eine exakte Bewertung von Spielfeldern mit acht oder weniger Steinen innerhalb weniger Millisekunden. Nach Verlassen des Eröffnungsbuches kann der – im Rahmen dieser Arbeit erstellte – Alpha-Beta-Agent praktisch alle Positionen in deutlich weniger als einer Sekunde analysieren. Dies ist für die meisten Zwecke völlig ausreichend. Lediglich einige Stellungen mit 9-12 Spielsteinen können zu einer etwas längeren Bedenkzeit führen (bis zu 10 Sekunden).

4.5.3 12-Ply Eröffnungsdatenbanken

Die im vorherigem Abschnitt betrachtete 8-Ply Eröffnungsdatenbank verfügt über einen entscheidenden Nachteil, der nicht sofort offensichtlich ist: Alle enthalten Spielstellungen verfügen zwar über eine Bewertung Sieg, Niederlage oder Unentschieden. Allerdings kann keine Aussage darüber gemacht werden, wie weit Siege oder Niederlagen von der aktuellen Position entfernt sind. Diese Distanzen sind aber insbesondere für den nachziehenden Spieler wichtig, um frühzeitige Niederlagen zu verhindern. So muss ein Alpha-Beta-Agent auch oft über die Grenzen des Eröffnungsbuches hinaus suchen, um schlechte Züge für den Nachziehenden zu vermeiden.

Für den Anziehenden könnten diese Informationen ebenfalls hilfreich sein. Obwohl dieser bei perfektem Spiel immer gewinnt, sollten frühzeitige Gewinne generell bevorzugt behandelt werden. Aus diesem Grund und zur Erreichung weiterer Laufzeitvorteile wurde sich dazu entschieden eine weitere Eröffnungsdatenbank – diesmal jedoch mit allen Spielstellungen mit 12 Spielsteinen – berechnen zu lassen.

Zu diesem Zweck wurden zunächst alle möglichen legalen 4.200.899 Stellungen ermittelt. Auch hier wurden identische Stellungen (gespiegelt an der mittleren Spalte)

nur in einer Variante übernommen. Dies führt zu einer deutlichen Einsparung an Speicherplatz und Rechenzeit.

Anschließend wurden die einzelnen Positionen bewertet und jeweils das theoretische Resultat inklusive Distanz zum Spielende ermittelt. Die Berechnung wurde auf einem Pentium4-Rechner ausgeführt und hat ca. drei Wochen in Anspruch genommen. Zur Validierung der erhalten Ergebnisse wurde aus der 12-Ply-Datenbank eine 8-Ply-Datenbank erzeugt und mit der bereits vorhandenen abgeglichen. Für einfache Stellungsbewertungen ohne die Notwendigkeit von Distanzangaben ist eine 12-Ply-Datenbank ausreichend, die nur Informationen zu Sieg / Niederlage oder Unentschieden der Stellungen enthält. Für diesen Fall steht auch ein eigenes Eröffnungsbuch zur Verfügung. Wie weiter unten beschrieben wird, benötigt diese weniger als 1/3 des ursprünglichen Speicherbedarfs.

Weiterhin wurde eine 10-Ply-Datenbank erstellt, die in diesem Projekt jedoch keine weitere Verwendung findet.

4.5.4 Codierung und Speicherung der Eröffnungsdatenbanken

Da die einzelnen Datenbanken viele Spielpositionen enthalten, muss eine geeignete Codierung der einzelnen Stellungen gefunden werden, die den benötigten Speicher minimiert. Hierzu kann die Tatsache ausgenutzt werden, dass alle Spielstellungen einer Datenbank die gleiche Anzahl an Spielsteinen enthalten, die (Anzahl) darüber hinaus vergleichsweise klein ist. Die Codierung eines Spielzustandes in 2 x 42 Bit (vgl. Unterkapitel 1.2) erscheint hier nicht sinnvoll, da alleine für die 12-Ply-Datenbank mit 4.200.899 Stellungen ca. 50 MB an Speicher notwendig wären.

Stattdessen soll hier eine Codierung betrachtet werden, die einer *Huffman-Codierung* entspricht und den Schwerkrafts-Charakter des Spieles ausnutzt. Folgende Beobachtung ist hierfür wichtig: Oberhalb eines leeren Feldes können sich nur noch weitere leere Felder befinden. Für eine eindeutige Beschreibung eines Spielzustandes reicht es daher, die Steine beider Spieler für jede Spalte in einer festgelegten Reihenfolge aufzulisten, ohne das die leeren Felder von Relevanz wären. Lediglich die Spalten müssen voneinander getrennt werden. Neben den beiden möglichen Spielsteinen, wird bloß noch ein *Trennzeichen* benötigt um eine eindeutige Codierung zu erreichen.

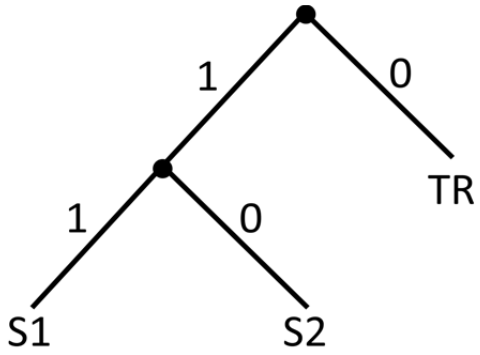

Abbildung 4.2: Huffman-Codierung der zwei möglichen Spielsteine und des Trennzeichens

Um eine Position zu codieren wird anschließend folgendermaßen vorgegangen: Die Spalten werden von links nach rechts, die Zellen innerhalb der Spalten von unten nach oben ausgewertet. Je nach Belegung der jeweiligen Felder wird das entsprechende Codewort der aktuellen Sequenz angehängt. Wird ein leeres Feld erreicht, so hängt man das Codewort für das Trennzeichen an die Sequenz an und fährt mit der nächsten Spalte fort. Sind alle Spalten durchlaufen, ist für die letzte kein Trennzeichen mehr nötig und die Sequenz ist vollständig. Insgesamt sind also sechs Trennzeichen und vier bzw. sechs Steine für die beiden Spieler notwendig. Aus diesem Grund wird dem Trennzeichen das Codewort der Länge eins zugeordnet.

Der Spieler am Zug muss nicht codiert werden, da dieser eindeutig aus der Position selbst ermittelt werden kann, der Spielzustand ist daher vollständig durch die Stellung beschrieben. Ist die Position codiert, wird die Bewertung der Stellung einfachheitshalber an die Sequenz angehängt. Sind keine Distanzen zum Spielende enthalten, reichen zwei Bit aus um die Position mit Sieg / Niederlage des Anziehenden oder Unentschieden zu bewerten. Dadurch kann eine Stellung mit acht bzw. zwölf Steinen inklusive Bewertung in genau drei bzw. vier Bytes codiert werden. In Abbildung 4.3 wird eine Codierung beispielhaft für eine Position mit acht Steinen durchgeführt.

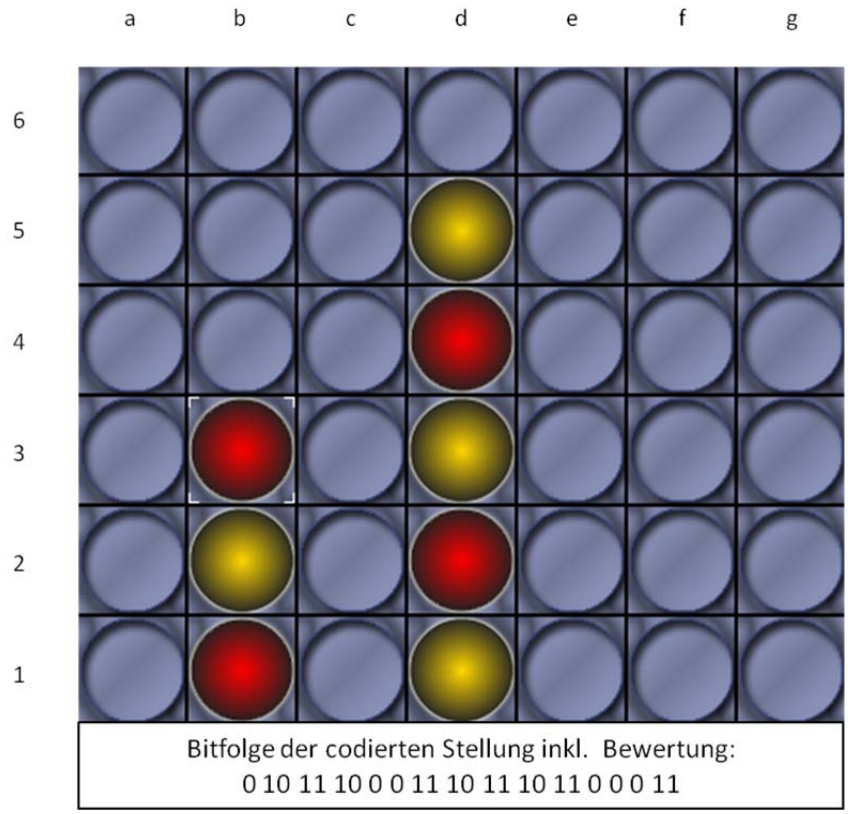

Abbildung 4.3: Codierung einer Spielstellung mit acht Steinen: Es werden 24 Bit (3 Byte) benötigt. Die letzten beiden Bits stellen die Bewertung der Stellung dar. In diesem Fall ist ein Sieg des Anziehenden im weiteren Spielverlauf zu erwarten.

Für die 12-Ply-Datenbank mit den jeweiligen Distanzen zum Spielende werden fünf Bytes reserviert, wobei das fünfte Byte die Bewertung enthält. Folgende Darstellung wird hierzu verwendet:

$$v_i = \begin{cases} +100 - Distanz, \text{für } einen\ Sieg\ des\ Anziehenden \\ -100 + Distanz, \text{für } einen\ Sieg\ des\ Nachziehenden \\ 0, sonst \end{cases} \quad (4.2)$$

wobei die Distanz die theoretische Entfernung zum Spielende von der aktuellen Stellung aus (bestehend aus 12 Spielsteinen) angibt.

4.5.5 Komprimierung und Sortierung der Eröffnungsdatenbanken

Wird die Tatsache ausgenutzt, dass alle Datenbanken vollständig sind – daher alle nötigen legalen Positionen enthalten – können die Datenbanken mit den einfachen Stellungsbewertungen (Sieg / Niederlage / Unentschieden) noch weiter verkleinert werden. Entfernt man alle Spielstellungen mit einem voraussichtlichem Sieg des anziehenden Spielers aus der 8- bzw. 12-Ply-Datenbank – die immerhin knapp 60% der Stellungen in beiden Datenbanken ausmachen – kann nach wie vor die exakte Bewertung einer Position erfolgen. Taucht eine Stellung nicht in der Datenbank auf, ist von einem Sieg des Anziehenden auszugehen. Allerdings ist auch die gespiegelte Variante der Stellung mit zu berücksichtigen.

Durch diesen Schritt verkleinert sich die 8-Ply-Datenbank von 86.893 (250 kB) auf 34.286 Spielstellungen (100 kB). Die neue 12-Ply-Datenbank enthält nun noch 1.735.945 Stellungen (6,6 MB) von ursprünglich 4.200.899 (16,0 MB).

Da in der 12-Ply-Datenbank mit den exakten Gewinndistanzen auf keine Spielstellung verzichtet werden kann (ansonsten würden die Distanz-Informationen verloren gehen), werden darin weiterhin alle 4.200.899 Stellungen gehalten, die einer Größe von 20,0 MB entsprechen.

Aufgrund dessen, dass unter Umständen sehr viele Zugriffe auf die berechneten Datenbanken in kürzester Zeit erfolgen könnten, sollten diese zur Laufzeit in den Arbeitsspeicher geladen werden. Um die Suche innerhalb der Datenbanken zu beschleunigen, wurden diese sortiert, sodass man die *Binäre Suche* einsetzten kann. Dadurch kann die Komplexität der Suche auf $O\big(log(n)\big)$ reduziert werden, was eine deutliche Steigerung im Vergleich zur linearen Suche bedeutet. In vielen Fällen muss die gespiegelte Variante einer Position in der Datenbank gesucht werden, infolgedessen ergibt sich pro Stellung eine max. Anzahl an Zugriffen von ca. 2x 15 (8-Ply-Datenbank), 2x21 (12-Ply-Datenbak ohne Gewinn-Distanzen) und 2x 22 (12-Ply-Datenbak mit exakten Gewinn-Distanzen).

4.6 Bestimmung der spieltheoretischen Werte innerer Knoten

Viele Situationen des *Vier Gewinnt* Spiels erlauben es einem Spieler den Gegner unmittelbar in eine ausweglose Situation zu bringen, sodass das Spiel bereits im nächsten Zug gewonnen werden kann. Dies ist mit der Bildung einer Zwickmühle während eines Mühlespiels vergleichbar. Besitzt einer der beiden Spieler eine Zwickmühle, kann dieser durch Schließen einer Mühle gleichzeitig eine weitere öffnen, sodass sich die Spielsteine des Gegners oft sehr schnell dezimieren lassen. Allerdings bedeutet die Erzeugung einer Zwickmühle, anders als bei *Vier Gewinnt*, nicht die sofortige Niederlage des Gegners. Typischerweise lassen sich beim *Vier Gewinnt* Spiel drei Arten von ausweglosen Stellungen unterscheiden[24]:

1. Zum Spielende hin treten häufig Situationen auf, in denen einer der Spieler nur noch unter die Drohung des Gegners ziehen kann und das Spiel anschließend verliert (Zugzwang).
2. In gewissen Fällen gelingt es einem der Opponenten zwei unmittelbare Drohungen mit einem Halbzug zu erzeugen. Der Gegner kann jedoch nur eine dieser beiden unmittelbaren Drohungen neutralisieren und hat damit verloren.
3. Enthält eine Spalte zwei Drohungen eines Spielers – die gleich übereinander liegen – muss der betreffende Spieler lediglich unter die erste Drohung ziehen, um das Spiel zu gewinnen.

Da sich der erste Punkt ohnehin auf die *Blattknoten* bezieht, ist an dieser Stelle ausschließlich die Betrachtung der beiden letzten Punkte von Interesse. Solche Situationen können von der einfachen Suche oft erst zu spät erkannt werden, da die Zugsortierung häufig zunächst andere Züge bevorzugt. Aus diesem Grund wird in dem Programm, noch vor dem Expandieren der einzelnen Teilbäume, jeder Zug darauf geprüft, ob dieser eine Situation nach Punkt 2. oder 3. herstellen kann. Ist dies der Fall, muss der betreffende Knoten nicht weiter untersucht werden, da eine exakte Bewertung der Stellung bereits vorliegt. Die Untersuchung von Positionen nach den o.g. Kriterien ist verhältnismäßig laufzeitintensiv, sodass ein Einsatz bloß innerhalb wurzelnaher Knoten in Frage kommt. Dennoch kann mithilfe dieser Maßnahme die vollständige Analyse zahlreicher Teilbäume vermieden und die Suche somit wesentlich beschleunigt (ca. 20 %) werden.

Eine weitere, wenn auch minimale, Verbesserung (< 2%) wird erreicht, indem man die Suche bereits immer dann abbricht, wenn 40 Spielsteine auf dem Brett enthalten sind, da bereits zu diesem Zeitpunkt eine exakte Aussage über den Spielausgang möglich ist.

[24] Der zweite und dritte Punkt sind jedoch nur unter der Voraussetzung realisierbar, dass der Gegner nicht nach dem Zug des aktuellen Spielers gewinnen kann.

4.7 Evaluierung eines Spielzustandes am Suchhorizont

Stellungen für einfache Spiele wie *Tic Tac Toe* lassen sich mithilfe einer Baumsuche ohne besondere Probleme analysieren, da der Spielbaum bis zu den Blättern entwickelt werden kann.

Bei einer deutlich größeren Komplexität, wie dies z.B. bei *Vier Gewinnt* der Fall ist, ist dies nicht mehr ohne weiteres möglich. Hier muss man die Suchtiefe oft auf einen akzeptablen Wert begrenzen. Erreicht die Suche den Horizont, ist eine Funktion nötig, die den spieltheoretischen Wert des aktuellen Spielzustandes einschätzen kann. Die Einschätzung ist aber in der Regel nicht trivial, da einerseits eine hohe Qualität der *Evaluierungsfunktion* erwartet wird. Andererseits muss dies mit einer vertretbaren Laufzeit geschehen, um eine weitere unnötige Begrenzung der Suchtiefe zu vermeiden. Zur Lösung dieses Zielkonfliktes ist ein hohes Maß an Kreativität und sehr detailliertes Wissen über das betreffende Spiel nötig, damit die relevanten Merkmale einer Stellung identifiziert und geeignet bewertet werden können.

Im Gegensatz zu *Schach* ist es möglich das Spiel *Vier Gewinnt*, wenn auch mit einem sehr großen Programmieraufwand, bis zu den terminalen Blattknoten des Spielbaumes zu analysieren. Die dazu eingesetzten Techniken und Hilfsmittel sind in den vorangegangen Unterkapiteln ausführlich beschrieben worden.

Daher wird in dieser Arbeit kein besonderer Wert auf die statische Stellungsbewertung am Suchhorizont gelegt, da lediglich eine perfekte Baumsuche als Vergleichsmöglichkeit für das TD-Learning erforderlich ist. Nichtsdestotrotz sollen kurz einige Ideen beschrieben werden, aus denen eine kompakte Evaluierungsfunktion resultierte.

Zunächst ist die Identifizierung der zu bewertenden Merkmale nötig. Während beim *Schach* das Materialverhältnis eine wichtige Rolle spielt, ist für *Vier Gewinnt* vor allem die Anzahl der Drohungen beider Spieler und deren Anordnung von Bedeutung. Weitere Merkmale, wie z.B. die Positionierung einzelner Steine, sind zwar denkbar, wurden im Rahmen dieser Arbeit jedoch nicht weiter untersucht.

Im ersten Teil der Stellungsbewertung werden daher zunächst alle Drohungen beider Spieler ermittelt. Anschließend untersucht eine Heuristik die Anordnung der Drohungen und schätzt den voraussichtlichen spieltheoretischen Wert. Für Spielstellungen mit weniger als drei Drohungen kann dieser Schritt mithilfe von wenigen bitweisen Operationen und einigen Verzweigungen sehr schnell ausgeführt werden. Dies trifft auf ca. 70% der untersuchten Spielstellung zu. Sind allerdings mehr Drohungen vorhanden, ist ein höherer Aufwand zur Bestimmung einer Prognose notwendig. Details hierzu können dem Quelltext (*c4.AlphaBetaAgent.evaluate*) entnommen werden.

4.8 Verifizierung und Laufzeitmessung

Im Laufe des Projektes hat die Alpha-Beta Suche mit all ihren Erweiterungen und Verbesserungen einen enormen Umfang erreicht. Zum Zeitpunkt der Dokumentation umfasst der Quelltext nahezu 7000 Codezeilen, wobei einige Teile maschinell erzeugt wurden. Viele der eingesetzten Methoden sind auf spezielle Situationen während der Suche beschränkt. Die Korrektheit der Suche im Zusammenspiel mit den zusätzlichen Verfahren und Techniken lässt sich daher nicht ohne weiteres überprüfen. Die Untersuchung einiger weniger Stellungen ist hierfür nicht aussagekräftig genug.

Weiterhin können sich Änderungen am Quelltext in vielen Situationen positiv auf das Laufzeitverhalten auswirken, in anderen jedoch nicht. Häufig ist daher eine Aussage darüber, ob eine Modifikation tatsächlich eine Beschleunigung bewirkt, nicht generell möglich, auch nicht dann, wenn deutliche Knoteneinsparungen zu beobachten sind. Aus diesem Grund wurde sich dazu entschieden umfangreichere Tests durchzuführen. Da die 8-Ply Eröffnungsdatenbank von *John Tromp* (siehe Abschnitt 4.5.2) unabhängig von diesem Projekt erzeugt wurde, eignet sich diese besonders als Vergleichsmöglichkeit. Während der Überprüfung entnimmt die Testroutine der Datenbank in jedem Durchgang jeweils 2000 Einträge und gleicht diese mit den Resultaten der Alpha-Beta-Suche (die selbstverständlich keine Eröffnungsbücher nutzt) ab. Wird gleichzeitig auch die Laufzeit der Suche gemessen, verwendet die Testroutine für jeden Durchgang die gleichen Datenbank-Einträge, um Vergleiche möglich zu machen. In der aktuellen Version dauert die Suche auf einem Pentium4-Rechner durchschnittlich ca. 1,7 Sekunden für Positionen bestehend aus acht Spielsteinen (Größe der Transpositionstabellen: 25 MB).

Für die Suche vom leeren Spielfeld aus sind ungefähr 3,5 Minuten nötig, um zu erkennen, dass der Zug in die mittlere Spalte der bestmögliche ist (Größe der Transpositionstabellen: 50 MB). Dies scheint eine vergleichsweise gute Zeit zu sein. *Mustrum*[25], eines der bekanntesten *Vier Gewinnt* Programme, benötigt ohne Zuhilfenahme von Eröffnungsbüchern und mit einer 60 MB Hash-Tabelle für diese Erkenntnis fast 17 Stunden.

[25] Kann von www.mustrum.de heruntergeladen werden.

Literaturverzeichnis

[1] Victor Allis (1988): A Knowledge-based Approach of Connect-Four. The Game is Solved: White Wins. *Masters Thesis*. Department of Mathematics and Computer Science Vrije Universiteit Amsterdam.
www.connectfour.net/Files/connect4.pdf

[2] John Tromp: John's Connect Four Playground (abgerufen am 9. Januar 2012).
http://homepages.cwi.nl/~tromp/c4/c4.html

[3] Victor Allis (1994): Searching for Solutions in Games and Artificial Intelligence. *PhD thesis*. Department of Computer Science, University of Limburg; 165-166.
http://fragrieu.free.fr/SearchingForSolutions.pdf

[4] Simon M. Lucas (2008): Learning to Play Othello with N-Tuple Systems. Department of Computer Science, University of Essex, Colchester.
http://algoval.essex.ac.uk/rep/games/NTOthello/NTupleOthello.pdf

[5] Wolfgang Konen (2008): Reinforcement Learning für Brettspiele: Der Temporal Difference Algorithmus. *Technical Report*, Institut für Informatik, Fachhochschule Köln, Gummersbach.
http://www.gm.fh-koeln.de/~konen/Publikationen/TR_TDlambda.pdf

[6] Wolfgang Konen und Thomas Bartz-Beielstein (2008): Reinforcement Learning: Insights from Interesting Failures in Parameter Selection. Institut für Informatik, Fachhochschule Köln, Gummersbach.
http://www.gm.fh-koeln.de/~konen/Publikationen/ppsn2008.pdf

[7] Bruce Moreland: Fail Low, Fail High. Stand: 24. März 2001 (abgerufen am 9. Januar 2012).
http://web.archive.org/web/20040512194831/brucemo.com/compchess/programming/glossary.htm

[8] Hendrik Baier (2006): Der Alpha-Beta-Algorithmus und Erweiterungen bei Vier Gewinnt. *Bachelor-Thesis*. TU Darmstadt, Knowledge Engineering Group; 34-36, 46-48
http://www.ke.informatik.tu-darmstadt.de/lehre/arbeiten/bachelor/2006/Baier_Hendrik.pdf

[9] Albert L. Zobrist (1970): A new hashing method with application for game playing. *Technical Report 88*. Computer Sciences Department, The University of Wisconsin, Madison.
research.cs.wisc.edu/techreports/1970/TR88.pdf

[10] Aske Plaat u.a. (1994): Nearly Optimal Minimax Tree Search? *Technical Report 94-19*. Department of Computing Science, University of Alberta, Edmonton, Alberta; 9-13
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.7740