Reinforcement Learning in der Schachprogrammierung

Studienarbeit
(1.überarbeitete Fassung)
Freie Universität Berlin

Autor: ***Marco Block***
Dozent: ***Prof. Dr. Raúl Rojas***

16. September 2003

Inhaltsverzeichnis

Vorüberlegungen und Motivation

Das Schachspiel ist eines der Leitprojekte der Künstlichen Intelligenz und wird oft auch als „**drosophila of AI**“ bezeichnet. Zum Einsatz sind zahlreiche Algorithmen und Lernmethoden gekommen. Doch noch immer übersteigt die Komplexität des Schachs jene leistungsfähigen Rechner dieser Zeit, um das Schachspiel gänzlich berechenbar zu machen. Zur Veranschaulichung des Komplexitätsgrades soll Tabelle 1 dienen [Friedel]:

Halbzugtiefe	Anzahl der Schachstellungen	Stellungen/Sek bei 3 min/Zug
1	40	-
2	1600	$1,1$
3	64.000	7
4	$2,6$ Millionen	44
5	102 Millionen	281
6	$4,1$ Milliarden	1777
7	164 Milliarden	11.000
8	$6,5$ Billionen	70.000
9	262 Billionen	450.000
10	10 Billiarden	$2,8$ Millionen
11	419 Billiarden	17 Millionen
12	17 Trillionen	113 Millionen
13	671 Trillionen	719 Millionen
14	$2,7 * 10^{22}$	$4,5$ Milliarden

Tabelle 1: Anzahl der Stellungen in Abhängigkeit zu der Halbzugtiefe

Wie man der Tabelle 1 sehr eindrucksvoll entnehmen kann, müssten bei einer Halbzugtiefe von 14 und einer Zeitvorgabe von 3 Minuten pro Zug, $4,5$ Milliarden Stellungen pro Sekunde analysiert und ausgewertet werden. Das ist mit der heutigen Technologie nicht in zumutbarer Zeit erreichbar. Um dem Computer aber trotzdem vernünftiges Schachspielen beizubringen, bedarf es algorithmischen und schachtheoretischen Grundlagen. Einsparungen in der Vorausberechnung lassen sich sowohl bei Zugumstellungen, als auch Stellungen, deren Verlauf (aus schachlicher Sicht) schlecht sind finden.

Heutige Programme sind in der Lage den amtierenden Weltmeister zu schlagen. Ein sehr starkes Programm, welches Grossmeisterniveau erreicht hat, trägt den Namen ***KnightCap***. Eigentlich ist es nicht besonders sensationell ein Schachprogramm mit dieser Spielstärke zu schreiben , aber eine Besonderheit hat das Programm schon. Vom Erfolg Tesauros Backgammon-Programmes beeindruckt, welches das Spielen erlernt und stärker als der Weltmeister spielt, haben Jonathan Baxter, Andrew Tridgell und Lex Weaver versucht die gleiche Lernstrategie in ein Schachprogramm umzusetzen. Der erstaunliche Erfolg mit dem Einsatz von Reinforcement Learning in der Schachprogrammierung, wie er im letzten Abschnitt der Studie geschildert wird, brachte eine alte Idee in ein ganz neues Licht.

Anregungen und Kritik an:

Marco Block - block@inf.fu-berlin.de

Kapitel 1

Reinforcement Learning

Reinforcement Learning (RL) ist eine allgemeine Lernstrategie, die nicht durch bestimmte Lernalgorithmen definiert wird, sondern durch Aktionen innerhalb einer Umgebung und deren Reaktion darauf. Es gibt zwei spezielle Charakteristika: die **Suche durch Ausprobieren** und eine **verzögerte, positive Verstärkung**.

Oft findet man RL unter dem Oberbegriff *unüberwachtes Lernen*. Das unüberwachte Lernen charakterisiert Lernstrategien, die keinen *Lehrer* benötigen, der dem *Agenten*[1] Ausgaben vorgibt. Beim *überwachten Lernen*, setzt ein Lehrer Beispiele ein, um den Agenten zu korrigieren und damit zu trainieren. Beim RL findet der lernende Agent selbst durch Ausprobieren und Rückmeldungen (*Reinforcement Signal*) die optimale Strategie, um Ziele innerhalb seiner Umgebung zu erreichen (die Ziele werden vom Lehrer nicht vorgegeben, sondern lediglich evaluiert). Deswegen kann man auch sagen: „... der Lerner lernt nicht mit Lehrer, sondern mit einem Kritiker“[Wellner]. Daher sollte man RL als eigene Klassifizierung beschreiben, dem halbüberwachten [Zhang] oder bestärkenden Lernen, da es sich von den anderen unüberwachten Lernstrategien dahingehend unterscheidet, dass wir Einfluss auf die Bewertung durch ein Rückmeldesignal nehmen. Typischerweise wird aber beim unüberwachten Lernen kein Einfluss auf die Bewertung während der Trainingsphase genommen.

Es werden im weiteren Verlauf der Arbeit folgende Definitionen benötigt:

t diskreter Zeitpunkt im Problemlösungsprozess

s_t Problemzustand zum Zeitpunkt t, abhängig vom vorhergehenden Zustand s_{t-1} und der vorhergehenden Aktion a_{t-1}

a_t Aktion zum Zeitpunkt t, die abhängig von dem aktuellen Zustand s_t ist

r_t Belohnung zum Zeitpunkt t (Reward-Funktion), abhängig vom vorhergehenden Zustand s_{t-1} und der vorhergehenden Aktion a_{t-1}

π Handlungsstrategie für einen Zustand, folglich ist π die Zuordnung von Zuständen zu Aktionen (policy)

π^* optimale Strategie (unser Ziel)

V ordnet jedem Zustand s einen Wert zu, folglich erhalten wir bei Einsatz der Strategie π im Zustand s den Wert $V^\pi(s)$

α Lernrate

RL ist erfolgreich, wenn nicht nur das gegenwärtig Bekannte, sondern auch eine Weiterforschung der Welt verwendet wird.

RL-Regel 1 *Agenten, die immer oder niemals weiterforschen, werden stets scheitern!*

Praktisch können alle Ansätze von Reinforcement Learning den folgenden drei Kategorien zugeordnet werden [Luger].

[1] Das lernende System.

Dynamische Programmierung berechnet Wertefunktionen, indem sie Werte von Nachfolgerzuständen auf Vorgängerzustände übertragen. Beim Einsatz werden die einzelnen Zustände anhand eines Modells der nächsten Zustandsverteilung systematisch nacheinander aktualisiert. Die Realisierung von RL durch Dynamische Programmierung basiert auf folgender Gleichung:

$$V^{\pi}(s) = \sum_a \pi(a|s) * \sum_{s'} \pi(s \rightarrow s'|a) * (R^a(s \rightarrow s') + \gamma(V^{\pi}(s')))$$

Monte Carlo Bei den Monte Carlo-Methoden gibt es ähnlich wie bei der *Temporalen Difference* eine update-Funktion, diese wird aber erst nach erreichen eines Endzustandes (beim Schach, wäre es das Partieende), angewandt. Bei Monte-Carlo-Methoden ist es nicht nötig, die gesamte *Umwelt* zu kennen, sondern lediglich Auszüge dieser (Sequenzen), die dann repräsentierend für die Umwelt, Wahrscheinlichkeiten der Zusammenhänge liefern.

TD Temporal Difference-Methoden lernen von mittels Beispielen berechneten Verläufen und modifizieren Werte zwischen einzelnen Zuständen. Eine spezielle Variante ist das *Q-Lernen*. Dabei ist Q eine Funktion von Zustand-Aktion-Paaren zu gelernten Werten und es gilt: $Q : (Zustand\,x\,Aktion) \rightarrow Wert$. Für das Q-Lernen über einen Schritt gilt, mit $c, \gamma = 1$ und $r_{t+1} =$ Belohnung bei s_{t+1}:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + c * [r_{t+1} + \gamma * max_a(Q(s_{t+1}, a_{t+1}) - Q(s_t, a_1))]$$

Für die Schachprogrammierung eignen sich TD-Methoden, da es zum einen nicht möglich ist, die komplette Umwelt (alle Schachbrettkonstellationen mit Übergängen) zu modellieren und zum anderen eine Evaluationsfunktion existiert, die es erlaubt verschiedene Zustände (zeitlich) in Relation zu setzen. Zunächst konstruiert man ein fast vollkommenes Schachprogramm, das dem Standard der heutigen Zeit in punkto Zugwahl und Stellungsbewertungsfaktoren im nichts nachsteht. Anschliessend hat man das Problem, die Evaluation mit „Leben“ zu füllen. Die Koeffizienten der Bewertungsfaktoren müssen eingestellt werden. Da je nach Programmiertechnik und Anzahl der Faktoren die Bewertung von Programm zu Programm nicht einheitlich sein kann, muss nach jedem Bau eines Schachmotors die Justierung in langen Testphasen vorgenommen werden. Mit dem TD-Algorithmus haben wir eine gute Methode zur Verfügung, die Parameter durch „try and error“ selbst zu justieren. Wir konstruieren einen Agenten, der sich an das einfache Ziel „Schachpartien zu gewinnen“ hält.

1.1 Beispiel: Tic-Tac-Toe

Um ein besseres Verständnis für das Konzept Reinforcement Learning zu erlangen, soll nun das oft zitierte Tic-Tac-Toe-Beispiel dienen. Dazu konstruieren wir eine Wertetabelle (*value-function-table*), deren Einträge für jede Position einer Stellung und deren Positionsbewertung (*state current value*) stehen. Sollte der Wert eines Zustands A grösser als der eines Zustands B sein, so bedeutet das, dass die Gewinn-Wahrscheinlichkeit von A grösser als die von B ist. Um keinerlei Vorabinformationen zu liefern, werden bei der Initialisierung alle Werte auf 0.5 gesetzt (0 für Kreis und 1 für Kreuz gewinnt).

Nun beginnt der Agent zu spielen. Um einen Zug zu wählen schaut er in die Wertetabelle und wird zumeist „gierig“ den höchtbewerteten (besten) wählen. Damit erhöhen sich die Gewinnchancen. Nach der **RL-Regel 1** soll er aber auch forschende Züge unternehmen, dazu muss er auch manchmal zufällige Wege wählen, um Zustände zu besuchen (und damit zu bewerten), die er sonst nie betrachtet hätte.

Nun ist aber gerade der Vorteil vom RL, dass beim Übergang in einen neuen Zustand, ein alter angepasst werden kann. Dazu werden die Positionsbewertungen aktualisiert. Sei s_t der Zustand vor dem greedy-Move[2] und s_{t+1} der Zustand danach, dann ist der angepasste Wert $V(s)$.

$$V(s_t) \rightarrow V(s_t) + \alpha * [V(s_{t+1}) - V(s_t))],$$

[2] greedy = „gierig“; der vielversprechenste (momentan beste) Zug wird gewählt

wobei α ein kleiner positiver Wert ist (*step-size-parameter*), der die Lernrate bestimmt. Die Aktualisierungsregel (*updatefunction*) der Temporalen Differenz ist $V(s_{t+1}) - V(s_t)$, da die Angleichung auf eine Abschätzung zu zwei verschiedenen Zeiten basiert. Wenn man sich vorstellt, dass der α-Wert mit der Zeit kleiner wird, dann konvergiert diese Methode zu einem Agenten, der optimal spielt, oder anders gesagt, konvergiert diese Methode zu einer optimalen Strategie. Um nun eine gute Strategie zu erhalten müssen sehr viele Spiele gemacht werden und α langsam gegen 0 konvergieren.

RL-Regel 2 *Die Lernrate α, sollte sehr langsam gegen 0 konvergieren.*

Sollte α nicht gegen 0 konvergieren, sondern bei einem kleinen Wert (der nur noch wenig Einfluss auf die Aktualisierung besitzt) stehen bleiben, so besitzt das Programm trotzdem eine gute Strategie, aber verändert diese immer noch ganz langsam von Spiel zu Spiel. Wichtig ist noch zu sagen, dass es nötig ist, dem Agenten keinerlei Vorabinformationen zu geben.

Da **KnightCap** auf einer Lernstrategie basiert, die Temporale Differenz verwendet, wird im folgenden Abschnitt noch etwas tiefer darauf eingegangen.

1.2 Temporale Differenz

Algorithmen der Klasse *Temporale Differenz* lernen, indem sie die Diskrepanz zwischen den Zuständen des Agenten zu verschiedenen Zeitpunkten verringern. Dabei ist TD eine Kombination aus der Monte-Carlo-Idee und dynamischer Programmierung. Wie Monte-Carlo-Verfahren kann das TD-Verfahren direkt mit grober Erfahrung ohne ein Modell der Umweltdynamik lernen. Der Vorteil von TD ist aber, dass nicht wie bei Monte Carlo der Zyklus einmal durchlaufen werden muss, um die Parameter aktualisieren zu können. Es kann schon im nächsten Zustand eine Aktualisierung des vorhergehenden vollzogen werden. Trotzdem konvergiert TD zu einer optimalen Strategie. Wie Dynamische Programmierung, updated TD die Zustandswerte anhand bereits gelernter Zustände, ohne zu wissen, wie das Ziel aussieht. Ein Vorteil von der Temporalen Differenz ist es, dass keine Umwelt erforderlich ist.

Kapitel 2

Grundlagen der Schachprogrammierung

2.1 Grundbauplan

Der Schachmotor - das Herzstück eines Schachprogrammes - besteht nur aus der Berechnungseinheit. Die Verwaltung der Oberfläche und Module, wie eine Eröffnungsdatenbank gehören nicht dazu. Im Prinzip soll der Schachmotor nach einer Zugeingabe und anschliessender Verwendung eines Algorithmus einen gültigen, möglichst sehr guten Antwortzug liefern. Dazu muss der Motor die Figurenkonstellation intern repräsentieren. Aus schachlicher Sicht ist es auch wichtig, den Spielverlauf bis zum aktuellen Zug zu kennen (zum Beispiel, wenn zu bestimmen ist, ob eine Rochadeerlaubnis vorliegt). Damit nun der Schachmotor keinen zufälligen Zug zurückliefert, sollte ein entsprechender Algorithmus den besten nach bestimmten Kriterien ermitteln.

2.1.1 Brett-Repräsentation

Ein sehr wichtiger Teil der Schachprogrammierung ist die Wahl der Brett-Repräsentation. Wie sollen die Figuren auf dem Brett gespeichert und später durch geeignete Funktionen, wie gibFigur(x,y) oder setzteFigur(x,y) verwaltet werden? Wie einfach lassen sich anhand der Darstellungwahl (es gibt sehr viel Möglichkeiten) Züge generieren? Eine Möglichkeit der Brettdarstellung wäre es eine 8x8-Matrix zu nehmen (Felder) und jedes Feld mit einer Zahl zu kodieren. Z.B. eine 0 für ein leeres Feld, positive Zahlen für weisse, bzw. negative für schwarze Figuren (Tabelle 2.1).

-4	-2	-3	-5	-6	-3	-2	-4
-1	-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
4	2	3	5	6	3	2	4

Tabelle 2.1: 8x8 - Brettdarstellung

Als Standardverfahren hat sich eine Technologie etabliert, die sich *BitBoard* nennt. Dabei wird das Brett in verschiedenen 64-Bit Worten gespeichert. Es gibt z.B. eines für jeden Figurentyp und eines für die Farben (die Gesamtanzahl der *BitWorte* hängt von der Komplexität des Schachprogrammes ab). Nehmen wir als Beispiel ein BitWort indem ein schwarzer Springer auf dem Feld d3 steht und ein weiteres BitWort, mit

den weissen Bauern in der Ausgangsposition. Durch eine UND-Verknüpfung dieser zwei BitWorte lässt sich nun leicht[1] ermitteln, welche Springer-Bauern-Schlagzüge möglich sind (Tabelle 2.2).

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	**1**	0	**1**	0	0	0
0	**1**	0	0	0	**1**	0	0
0	0	0	**X**	0	0	0	0
0	**1**	0	0	0	**1**	0	0
0	0	**1**	0	**1**	0	0	0

&

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	0	0	0	0	0	0	0

=

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	**1**	0	0	0	**1**	0	0
0	0	0	0	0	0	0	0

Tabelle 2.2: Springer-Bauern-Schlagzüge

Obwohl die BitBoardtechnologie als Standard erwähnt wird, unterscheiden sich die Realisierungen in den einzelnen Progammen erheblich. Es gibt auch eine Variante der BitBoards, die sogenannten RotatedBit-Boards, die durch einen Mehraufwand an Vorausberechnungen (vor der Partie werden zahlreiche BitWorte vorab berechnet, speziell die Diagonalen) und grösseren Speicherbedarf aber eine schnelle Zuggenerierung ermöglichen. Vor- und Nachteile verschiedener Brettrepräsentationen werden auch im *FUSC#-Projekt*[2] der Freien Universität Berlin nachgegangen [Block].

ToPieces Board

Eine interessante und in vielerlei Hinsicht vorteilhafte Idee ist das in KnightCap verwendete ToPieces-Board. Jedes Feld des Schachbretts wird durch ein 32-Bit Wort repräsentiert. Die 32-Bit stehen für die 32 Startfiguren. Betrachtet man nun das Feld *j*, dann steht das *i*-te Bit für eine Figur *i*, die das Feld *j* attackiert. Durch diese Modellierung lassen sich komplizierte Faktoren der Stellungsbewertung, die sehr oft grosse Zeitreserven kosten, leicht beschreiben. Wir speichern also $64x32$ Bit. Die Brettfelder werden angefangen vom Feld a1 zeilenweise aufsteigend durchnummeriert. Aber jede gute Idee hat auch ihren Haken: bei dieser Repräsentation liegt das Problem in der MacheZug-, bzw. NehmeZugZurück-Methode. Denn es ist zeitlich aufwendig (und programmiertechnisch nicht sehr einfach realisierbar) die 32-Bit zu aktualisieren.

2.1.2 Zuggenerator

Nachdem wir uns für eine Brett-Repräsentation entschieden haben, müssen wir eine Zuggeneratorklasse schreiben, die bei Eingabe einer korrekten Stellung und der am Zug befindlichen Partei, eine Zugliste aller möglichen Züge zurückgibt. Dazu bieten verschiedenen Brettdarstellungen Vor- und Nachteile. In fast allen führenden Schachprogrammen werden BitBoards verwendet, da deren Einsatz mehr Vor- als Nachteile verspricht. Das Ziel der Schachmotorentwickler ist es immer noch die Berechnungszeit des Zuggenerators und der Stellungsbewertung zu minimieren, um tiefer rechnen zu können.

Ein Zuggenerator übernimmt auch die Funktion, Züge zu prüfen, die den König in eine Schachposition bringen und somit nicht legal sind. Ein Trick der dabei verwendet werden kann, - auf die Königposition wird eine Art Superfigur gesetzt, die in alle Richtungen mit den Spezialzügen jedes Figurentyps ziehen kann. Somit könnte ein Springer auf der Königsposition einen gegnerischen Springer schlagen und entlarvt dadurch das Schachgebot dieses Springers. Der vorhergehende Zug war also entweder nicht regelkonform oder, wenn die mögliche Zugliste leer ist, dann steht der König im matt.

[1] Dieses Verfahren ist schnell (*Prozessorebene*) und meist (je nach Programmiersprache) in einer kurzen Zeile programmierbar.

[2] FUSC# ist ein in C# programmierter Schachmotor. Das Projekt wurde im Oktober 2002 gegründet und hatte sein Turnierdebüt zur Langen Nacht der Wissenschaften im Juni 2003 (**http://www.inf.fu-berlin.de/~fusch**)

2.1.3 Zugwahl

Die Zugwahl ist der Abschnitt, in dem entschieden wird mit welchem Algorithmus der Zugwahlbaum konstruiert werden soll. Dabei basiert jeder Zugwahlalgorithmus auf den MinMax-Algorithmus. Zwei Spieler sind abwechselnd am Zug und beide versuchen, die Vorteile der Stellung zu ihren Gunsten zu verändern. Je besser der Zugwahlalgorithmus arbeitet, desto weniger unnötige Stellungen werden betrachtet.

MinMax-Algorithmus

Der MinMax-Algorithmus stellt die Basis für alle Schachalgorithmen dar. Zunächst wird ein vollständiger Suchbaum der Tiefe t aus den jeweiligen Zugmöglichkeiten der beiden Spieler aufgebaut. Anschliessend werden die Blätter mit einer Stellungsbewertung evaluiert. (Der Suchbaum lässt sich auch rekursiv ermitteln, wie es im weiteren Abschnitt besprochen wird.)

Der Algorithmus geht davon aus, dass jede Partei den für sich besten Zug wählt. Von den Blättern aus können wir diesen dann an den Vaterknoten weiterreichen. Die Strategie von Weiss ist es, eine Stellung mit maximalem Wert zu erreichen, da ein positiver Wert einem weissen Vorteil entspricht. Schwarz strebt das Minimum an.

```
/* PseudoCode
   Eingabe: Berechnungstiefe
   Ausgabe: bestmögliche Bewertung für die Partei, die
            am Zug ist
*/
1 int MinMax(int tiefe) {
2    if (weissamzug) return MaxKnoten(tiefe) // Weiss maximiert
3    else return MinKnoten(tiefe)            // Schwarz minimiert
4}
```

Die MaxKnoten-Routine arbeitet alle möglichen Züge rekursiv ab und gibt den Wert des besten nach oben. Von der Wurzel aus ist nun der beste Zug identifiziert worden.

```
/* PseudoCode
   Eingabe: Berechnungstiefe
   Ausgabe: Wert der maximal erreichbaren Stellung
*/
 1 int MaxKnoten(int tiefe) {
 2    if (tiefe == 0) return Stellungsbewertung();
 3    int bestmöglich = -∞;
 4    zugliste = GeneriereAlleZüge();
 5    while (zugliste_nicht_leer) {
 6       MacheNächstenZug();
 7       wert = MinKnoten(tiefe-1);
 8       NimmZugZurück();
 9       if (wert>bestmöglich) bestmöglich=wert;
10    }
11    return bestmöglich;
12 }
```

Die MinKnoten-Routine versucht nun für Schwarz die Stellung zu minimieren.

```
/* PseudoCode
   Eingabe: Berechnungstiefe
   Ausgabe: Wert der minimal erreichbaren Stellung
*/
 1 int MinKnoten(int tiefe) {
 2    if (tiefe == 0) return Stellungsbewertung();
 3    int bestmöglich = +∞;
 4    zugliste = GeneriereAlleZüge();
 5    while (zugliste_nicht_leer) {
 6       MacheNächstenZug();
 7       wert = MaxKnoten(tiefe-1);
```

```
 8       NimmZugZurück();
 9       if (wert<bestmöglich) bestmöglich=wert;
10    }
11    return bestmöglich;
12 }
```

Laufzeit- und Praxisanalyse:

Da der MinMax-Algorithmus einen vollständigen Baum der Tiefe t aufbaut, kann die Laufzeit mit O($t * log t$) abgeschätzt werden. Jede zusätzliche Halbzugebene ist um ein vielfaches grösser als der zuvor konstruierte Baum mit Tiefe $t-1$, da im Durchschnitt 40 Züge pro Stellung [Friedel] möglich sind.

Principal Variation (AlphaBeta-PV-Algorithmus)

Die Idee dieses Verfahrens ist es, Teilbäume, die einen bestimmten forcierten Weg im Baum durch korrekte Spielweise des Gegners nur noch verschlechtern könnte, nicht weiter zu berechnen (*AlphaBeta-Algorithmus*). Als Optimierung zu dem „Standard"-AlphaBeta-Algorithmus werden alle Züge (bis auf den ersten) in einem *minimalen Fenster*[3] betrachtet [Heinz].

```
/* PseudoCode
   Eingabe: Berechnungstiefe, Alpha, Beta
   Ausgabe: Wert maximal erreichbarer Stellung
*/

 1 int AlphaBetaPV(int tiefe, int alpha, int beta)
 2 {
 3    boolean PVgefunden=False;
 4    if (tiefe == 0) return StellungsBewertung();
 5    GenerierealleZüge();
 6    while (zügevorhanden)
 7    {
 8       MacheNächstenZug();
 9       if (PVgefunden) {
10          wert = -AlphaBeta(tiefe-1, -alpha-1,-alpha);
11          if (wert>alpha)and(wert<beta)
12             wert = AlphaBeta(tiefe-1, -beta, -alpha);
13       }
14       else wert = -AlphaBeta(tiefe-1, -beta, -alpha);
15       NimmZugZurück();
16       if (wert >= beta) return beta;
17       if (wert > alpha)
18       {
19          alpha=wert;
20          PVgefunden = True;
21       }
22    }
23    return alpha;
24 }
```

2.1.4 Stellungsbewertung

Hier unterscheidet sich das grosse von dem kleinen Schachprogramm. Tief können alle mehr oder weniger Rechnen, aber nach welchen Kriterien sollen nun schlechte Züge identifiziert und *ad acta* gelegt werden? Es reicht sicherlich nicht, allein die Materialkomponente zu nehmen und anhand derer zu entscheiden, welcher Stellung nun besser oder schlechter ist. Die Stellungsbewertung ist viel mehr. Hier werden sehr viele Faktoren bewertet und je nach Stellungtyp mehr oder weniger in die Bewertung aufgenommen. Beispielsweise ist es ratsam in der Eröffnungsphase die Rochade des eigenen Königs anzustreben (den König an der Brettecke in Sicherheit zu bringen). Im Endspiel ist es dagegen aus schachtheoretischer Sicht besser, den König zu zentrieren, denn nun stellt er eine aktive Spielfigur dar. Andererseits stellt sich die Frage der Effizienz. Es ist sicherlich besser einem Schachprogramm mehr als nur den Materialfaktor beizubringen, aber

[3] Damit sind nah beieinander liegende *alpha*- und *beta*-Werte beim rekursiven Aufruf vom Algorithmus gemeint.

auch nicht ratsam, (unnötig) kompliziert zu berechnende Faktoren einzubauen, deren Stellungsbewertung zeitlich dann so teuer wird, dass keine wirkliche Vorrausberechnung mehr möglich ist. Zu erwähnen ist noch, dass der zweitwichtigste Faktor bei der Schachprogrammierung nach dem Materialfaktor, der Mobilitätsfaktor ist. Dieser ist leicht zu berechnen und vereint eine grosse Anzahl schachtheoretischer Ansichten (z.B. bessere Figurenstellung). Berechnet wird dieser Faktor durch :

$$\#meine\,Zugmoeglichkeiten - \#gegnerische\,Zugmoeglichkeiten$$

Es ist sicherlich von Vorteil eine Stellung anzustreben, in der „mir“ mehr Züge zur Verfügung stehen, als meinem Gegner. Das Problem hierbei stellt aber die Rolle der Dame dar, da es nicht ratsam ist in der Eröffnungsphase die Dame möglichst zentral zu postieren um viele Felder zu überdecken (also viele Züge machen zu können), sondern sie ersteinmal nicht den Angriffen der gegnerischen Figuren auszusetzen, da diese dann schnell einen Entwicklungsvorsprung erzielen können. Demnach sollten zu Beginn die Damenzüge aussen vor gelassen werden.

2.1.5 Erweiterungen

Es gibt in der Schachprogrammierung zahlreiche leistungssteigernde Erweiterungen (*performance features*) einige davon habe ich hier aufgelistet, da sie für KnightCap verwendet werden.

Null Moves

In der Vorausberechnung kann es sein, dass ein bestimmter Schlagzug (z.B. Dame schlägt Bauer) aus schachlicher Sicht nicht gut ist[4] und dementsprechend untersucht wird, falls nun in zwei oder drei Zügen, ohne das der Gegner die Möglichkeit besitzt zu ziehen, kein entsprechender Vorteil gezogen werden kann, dieser Zug nicht weiter verfolgt wird. Das liefert eine Performance-Steigerung (gerade in der Ruhesuche) von ca. $10-20$ Prozent ($1-2$ Halbzüge tiefer, oder ca. $200-300$ Elo mehr [Friedel]).

Transpositionstabelle (oder HashTable)

Wenn in einem Suchbaum eine Stellung evaluiert wurde, was meistens sehr teuer ist, dann ist es sehr wahrscheinlich diese Stellung auch ein zweites mal im Suchbaum durch Zugumstellung zu erhalten. Man könnte sich diese Stellung und berechneten Evaluierungswert unter einem entsprechenden Schlüssel (*key*) in einer Hashtabelle speichern und dort das nächste mal „nachfragen“, ob denn der Schlüssel schon vorhanden ist oder nicht. Sollte er vorhanden sein, so fällt die erneute Berechnung der Evaluation weg und man liest einfach den Wert aus der Hashtabelle. Dieser Ansatz lässt sich auch z.B. mit Stellungen und den dazugehörigen möglichen Zügen machen (erspart Zuggenerierung). Es gibt sehr viele Ansätze und die Performance-Steigerung entspricht ca. $5-10$ Prozent (Verdoppelung der Suchgeschwindikeit, ca. $50-100$ Elo mehr [Friedel]).

Zugsortierung

Bei Alpha-Beta-Algorithmen ist die Zugreihenfolge geschwindigkeitsentschiedend. Es gibt einige heuristische Überlegungen, wie z.B. die *KillerMoves*. Wenn im Suchbaum auf einen bestimmten Zug ein guter gegnerischer Antwortzug kam und damit diesen „widerlegt hat“, dann ist es sehr wahrscheinlich, dass dieser auch andere Züge widerlegen kann. Demnach werden die besten Züge in einer KillerTable gespeichert, sortiert und die Zugliste darauf abgestimmt.

[4] Bei der Ruhesuche werden alle Schlag- und Schachzüge berechnet, um Stellungen genauer bewerten zu können.

Parallele Suche

Min-Max-Algorithmen lassen sich sehr unproblematisch parallel und dementsprechend durch Cluster-Systeme verarbeiten. Beispielsweise könnte ein Zug, der in der Zugliste steht und nun berechnet werden soll, von einem und der nächste von einem anderen Rechner verfolgt werden. Das Problem hierbei stellt lediglich ein organisatorisches dar. Knight Cap wurde so implementiert, dass es (auf Unix-Systemen) verteilt rechnen kann.

Kapitel 3

KnightCap

KnightCap ist ein Schachprogramm, geschrieben von Jonathan Baxter, Andrew Tridgell und Lex Weaver. Im folgenden Abschnitt möchte ich die Arbeit der drei Programmautoren näher betrachten und mich dabei auf die Publikation „Baxter, Tridgell, Weaver: *KnightCap: A chess program that learns by combining TD(λ) with minmax search.* Publication 1997“ beziehen.

Das Programm verbindet die Reinforcement-Learning-Strategie Temporale Differenz (TD(λ)-Algorithmus) mit der Spielbaumsuchtechnologie. Dabei wird die Bewertungsfunktion mit TDLeaf(λ)[1] in Kombination mit dem MinMax-Algorithmus erlernt. TDLeaf ist identisch mit TD, der Unterschied liegt lediglich darin, dass nicht auf den sich ergebenen Positionen (einer Partie) gearbeitet wird, sondern auf den den Blättern des Suchbaumes (*LeafNodes*). Das MinMax-Prinzip wird vom *MTD(f)-Algorithmus*[2], ausgehend von jeder Position verwirklicht. Beeindruckend ist, dass Knightcap mit einem Startrating von 1650[3] in nur 3 Tagen (308 Spiele) eine Stärke von 2150 erreichte. Um die Effektivität von Temporaler Differenz in einem Schachprogramm zu testen, wurde diese integriert, aber zunächst schien es nicht möglich den TD-Ansatz, wie er bei dem erfolgreichen TD-Gammon (von Tesauro) angewendet wird, auch im Schachspiel zu benutzen. Das Hauptproblem ist die Bewertungsfunktion. Verändert sie sich bei kleinen Stellungsveränderungen im Backgammon auch nur etwas (glatte Funktion), so ist die Bewertungsfunktion beim Schach sehr uneben (kleine Veränderungen können sehr grosse Veränderungen in der Bewertung hervorrufen). Der Einsatz von Neuronalen Netzen im Backgammon ist folglich sehr vielversprechend[4]. Die Betrachtung nur eines Zuges im Voraus ist im Schach nicht gut, da Schach zu taktisch ist. Es ist eine schnelle Bewertungsfunktion erforderlich (lineare Bewertungsfunktion). Im folgenden Abschnitt soll nun erläutert werden, wie es doch gelingen kann, durch Kombination eines Zugwahlalgorithmus und einer Bewertung gute Ergebnisse zu erreichen.

KnightCap besitzt eine speziell, reichhaltige Boardrepräsentation, die eine relativ schnelle Berechnung einer recht hochentwickelten positionellen Bewertung ermöglicht (KnightCap ist trotzdem 10 mal langsamer als Crafty[5] und 6000 mal langsamer als Deep Blue[6]).

Die lineare Stellungsbewertung von KnightCap wurde nun, durch Spielen auf einigen Internetservern (z.B. ICC), mit TDLeaf(λ) trainiert. Der Haupterfolg bestand darin, dass das Programm mit einer Bewertungsfunktion startete, deren Koeffizienten auf 0 gesetzt waren (lediglich die Materialwerte waren vorgegeben). TDLeaf wurde dazu genutzt, um durch das Internetspiel diese Koeffizienten zu aktualisieren. Im Anschluss an viele weitere Experimente wurde KnightCap mit einer verbesserten „Standardausrüstung“ versehen. Es wurde bespielsweise ein Eröffnungsbuch eingebaut.

[1] Eine Weiterentwicklung des TD(λ), um diesen auch für Schach verwenden zu können. [BaxTriWea2].
[2] Stark weiterentwickelte Variante des AlphaBeta-Algorithmus. Referenz zu finden bei [Heinz].
[3] 1650 entspricht einem gelegentlichem Vereinsspieler, 2150 einem Regionalspieler
[4] Es kann eine sehr aufwendige Bewertung stattfinden, da ein Zug nur mit der Tiefe 1 gewählt wird.
[5] Das stärkste „öffentliche“ Schachprogramm (Open Source Projekt).
[6] Spielte 1996 gegen Kasparov und gewann. Wurde anschliessend zerlegt.

3.1 Programmstruktur

KnightCap ist ein Schachprogramm, welches den aktuellen Standards entspricht. Als Brett-Repräsentation verwendet KnightCap ein ToPieces-Array und als Zugwahlalgorithmus den MTD(f). Zusätzlich verwendet KnightCap einige Heuristiken.

Als Standards, die mittlerweile in fast jedem Schachmotor zu finden sind, gehören auch hier die nullmoves, Asymmetrien, transposition tables und Zugsortierung (z.B. die Killer-Heuristik). Besonders zu erwähnen ist aber noch, das KnightCap für parallel arbeitende Rechner geschrieben wurde und dadurch grossen Vorteil ziehen kann (die Leistungstests fanden aber auf „Ein-Prozessor-Maschinen“ statt). Die Stellungsbewertung ist in 4 Stellungstypen untergliedert: Eröffnung, Mittelspiel, Endspiel und Mattstellungen. Jeder Stellungstyp verfügt über die gleichen Faktoren, die unterschiedlich bewertet sind. Eine Übersicht der wichtigsten Faktoren zeigt die Tabelle 3.1 [BaxTriWea2]

Stellungsfaktor	# Parameter		Stellungsfaktor	# Parameter	
BISHOP_PAIR	1		CASTLE_BONUS	1	
KNIGHT_OUTPOST	1		BISHOP_OUTPOST	1	
SUPPORTED_KNIGHT_OUTPOST	1		SUPPORTED_BISHOP_OUTPOST	1	
CONNECTED_ROOKS	1		SEVENTH_RANK_ROOKS	1	
OPPOSITE_BISHOPS	1		EARLY_QUEEN_MOVEMENT	1	
IOPENING_KING_ADVANCE	8		IMID_KING_ADVANCE	8	
IKING_PROXIMITY	8		ITRAPPED_STEP	8	
BLOCKED_KNIGHT	1		USELESS_PIECE	1	
DRAW_VALUE	1		NEAR_DRAW_VALUE	1	
NO_MATERIAL	1		MATING_POSITIONS	1	
IBISHOP_XRAY	5		IENDING_KPOS	8	
IROOK_POS	64		IKNIGHT_POS	64	
IPOS_BASE	64		IPOS_KINGSIDE	64	
IPOS_QUEENSIDE	64		IKNIGHT_MOBILITY	80	
IBISHOP_MOBILITY	80		IROOK_MOBILITY	80	
IQUEEN_MOBILITY	80		IKING_MOBILITY	80	
IKNIGHT_SMOBILITY	80		IBISHOP_SMOBILITY	80	
IROOK_SMOBILITY	80		IQUEEN_SMOBILITY	80	
IKING_SMOBILITY	80		IPIECE_VALUES	6	
THREAT	1		OPPONENTS_THREAT	1	
IOVERLOADED_PENALTY	15		IQ_KING_ATTACK_COMPUTER	8	
IQ_KING_ATTACK_OPPONENT	8		INOQ_KING_ATTACK_COMPUTER	8	
INOQ_KING_ATTACK_OPPONENT	8		QUEEN_FILE_SAFTY	1	
NOQUEEN_FILE_SAFETY	1		IPIECE_TRADE_BONUS	32	
IATTACK_VALUE	16		IPAWN_TRADE_BONUS	32	
UNSUPPORTED_PAWN	1		ADJACENT_PAWN	1	
IPASSED_PAWN_CONTROL	21		UNSTOPPABLE_PAWN	1	
DOUBLED_PAWN	1		WEAK_PAWN	1	
ODD_BISHOP_PAWN_POS	1		BLOCKED_PAWN	1	
KING_PASSED_PAWN_SUPPORT	1		PASSED_PAWN_ROOK_ATTACK	1	
PASSED_PAWN_ROOK_SUPPORT	1		BLOCKED_DPAWN	1	
BLOCKED_EPAWN	1		IPAWN_ADVANCE	7	
IPAWN_ADVANCE	7		IPAWN_ADVANCE2	7	
KING_PASSED_PAWN_DEFENCE	1		IPAWN_POS	64	
IPAWN_DEFENCE	12		ISOLATED_PAWN	1	
MEGA_WEAK_PAWN	1		IWEAK_PAWN_ATTACK_VALUE	8	

Tabelle 3.1: Stellungsfaktoren bei KnightCap

Eine aktuellere Version von KnightCap, die inzwischen mit einem lernenden Eröffnungsbuch ausgestattet ist, bewertet eine Stellung anhand von 5872 Funktionen (1468 für jeden der 4 Stellungstypen: Eröffnung, Mittelspiel, Endspiel und Mattstellung).

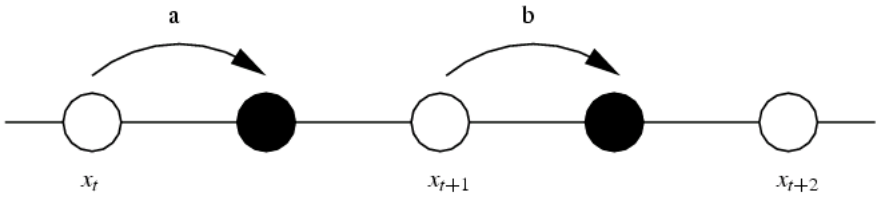
Damit hat es eine Spielstärke von $2400-2500$ (Spitze war 2575) und besiegt regelmässig internationale Meister. Das Problem bei einer so grossen Anzahl von Bewertungsfunktionen ist sicherlich der Verzicht auf eine tiefe Berechnung, da der Aufwand sehr gross ist. Qualitativ spielt das Programm, gerade in Blitzpartien aber sehr hoch.

3.2 TD-Leaf(λ)

TD-Leaf(λ) ist eine Variante des TD(λ)-Algorithmus und unterscheidet sich dahingehend, dass nicht auf allem Stellungen, die sich während der Partie ereignen, gearbeitet wird, sondern auf den Blättern des Suchbaum-Algorithmus. Hier findet auch immer eine Evaluation statt.

„Temporal Difference Learning“ wurde das erste mal beschrieben von A.L.Samuel (1959) und später erweitert und formalisiert von R.Sutton(1988). TD(λ) ist eine elegante Approximationstechnik. Die Parameter werden online, nach jedem Zustandsübergang oder möglicherweise in einem Stapel von Anpassungen, nach einigen Übergängen, angepasst.

Der TD-Algorithmus wird nun aus der Sicht eines „Agenten“ diskutiert.



Sei S die Menge aller möglichen Brettkombinationen. Spielvorgänge mit einer Zugserie zu einer diskreten Zeit werden schrittweise mit $t = 1,2,...$ angegeben. Zum Zeitpunkt t befindet sich der Agent auf dem Brett $x_t \in S$ und besitzt eine Liste von Zugmöglichkeiten (*actions*) A_{x_t} (als Aktion gilt, jeder legale Zug in der Stellung x_t). Der Agent wählt nun eine Aktion $a \in A_{x_t}$ aus und damit den Übergang des Zustands x_t zu x_{t+1} mit der Wahrscheinlichkeit $p(x_t, x_{t+1}, a)$. Hier ist x_{t+1} die Brettposition, nachdem der Agent einen Zug gemacht hat und der Gegner die Antwort gibt (Abbildung 3.1). Nachdem das Spiel zu Ende ist, bekommt der Agent einen Wert zurück (*Reward*), beispielsweise 0 für Remis, 1 für Sieg und -1 für eine Niederlage. Aus Bequemlichkeit nehmen wir an, dass jedes Spiel genau eine fixe Länge von N Zügen besitzt. Nun sei $r(x_N)$ der Antwortwert.

Betrachten wir die ideale Bewertungsfunktion $J^*(x)$.

$$J^*(x) := E_{x_N|x} r(x_N),$$

wobei das der Erwartungswert der Belohnung (*reward*) am Ende der Partie ist. Es ist schwer anzunehmen, dass jede Schachstellung durch eine lineare Bewertungsfunktion genau bewertet werden kann. Das Ziel ist nun, die ideale (wohl nicht lineare Bewertungsfunktion) $J^*(\bullet) : S \to \mathbb{R}$ durch eine lineare Funktion zu approximieren. Wir betrachten dazu eine parametrisierte Klasse von linearen Funktionen $J' : Sx\mathbb{R}^k \longrightarrow \mathbb{R}$. Gesucht ist $\omega = (\omega_1, ..., \omega_k)$ der Satz von Parametern, der die ideale Bewertungsfunktion $J^*(\bullet)$ möglichst gut approximiert. Der TD(λ)-Algorithmus tut nun genau das.

Sei $x_1, ..., x_{N-1}, x_N$ ist eine Zustandssequenz für eine gespielte Partie. Für einen gegebenen Vektor ω, ist die „temporale Difference“ von $x_t \to x_{t+1}$ definiert als

$$d_t := J'(x_{t+1}, \omega) - J'(x_t, \omega).$$

Man merke sich, dass d_t die Differenz zwischen dem Spielausgang von $J'(\bullet, \omega)$ zum Zeitpunkt $t+1$und dem Resultat von $J'(\bullet, \omega)$ zum Zeitpunkt t ist.

Für die ideale Bewertungsfunktion $J*$ gilt:

$$E_{x_{t+1}|x_t}[J^*(x_{t+1}) - J^*(x_t)] = 0,$$

mit der Idee: „Ich stehe besser, ergo erwarte ich einen Sieg.“. Aus Bequemlichkeit nehmen wir an, dass immer gilt: $J'(x_N, \omega) = r(x_N)$, - dann ergibt sich nun letztlich für die temporale Differenz

$$d_{N-1} = J'(x_N, \omega) - J'(x_{N-1}, \omega) = r(x_N) - J'(x_{N-1}, \omega).$$

d_{N-1} ist die Differenz zwischen dem wahren Ausgang der Partie (*reward*) und der Vorraussage zum Zeitpunkt des vorletzten Zuges. Am Ende des Spieles aktualisiert der TD(λ)-Algorithmus den Vektor w mit der Formel

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J'(x_t, \omega) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right],$$

wobei $\nabla J'(\bullet, w)$ der Gradient und $\left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right] =: \Delta t$ die gewichtete Summe der Differenzen im Rest der Partie ist. Für $\Delta t > 0$ gilt, dass die Stellung x_t vermutlich unterbewertet wurde, der Vektor w wird mit einem positiven Vielfachen des Gradienten addiert und demnach ist die Bewertung der Stellung mit den aktualisierten Parametern höher als zuvor. $\Delta t < 0$ interpretieren wir als Überbewertung der Stellung und addieren den Vektor mit einem negativen Vielfachen des Gradienten. Der positive Parameter α kontrolliert die Lernrate und konvergiert typischerweise (langsam) gegen 0. Der Parameter λ kontrolliert dabei den Umfang inwieweit sich die temporale Differenz zeitlich rückwärts fortpflanzt. Um das zu sehen schauen wir uns die vorhergehende Gleichung nochmal für $\lambda = 0$ an:

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J'(x_t, \omega) d_t$$

$$= \omega + \alpha \sum_{t=1}^{N-1} \nabla J'(x_t, \omega) \left[J'(x_{t+1}, \omega) - J'(x_t, \omega) \right]$$

und für $\lambda = 1$:

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J'(x_t, \omega) \left[r(x_N) - J'(x_t, \omega) \right]$$

Für $\lambda = 0$ wird der Parameter-Vektor so angepasst, dass $J'(x_t, \omega)$ näher an $J'(x_{t+1}, \omega)$ gesetzt wird. Wir beziehen uns also auf den nachfolgenden Zug. Im Kontrast dazu, TD(1) passt den Parameter-Vektor so an, dass die erwartete Belohnung zum Zeitpunkt t näher zum Resultat des Spieles zum Zeitpunkt N gesetzt wird. Wir beziehen uns direkt auf den Ausgang der Partie $(1, -1, 0)$ Ein λ-Wert zwischen 0 und 1 interpoliert zwischen diesen beiden Verhaltensweisen.

Wie lassen sich nun diese Erkenntnisse in einen Zug-Such-Algorithmus unterbringen? Eine Möglichkeit wäre es die gefundene Näherung $J'(\bullet, \omega)$ zu benutzen um aus einer grossen Anzahl von Aktionen zu einer Stellung den vielversprechensten zu wählen. Dabei wird die Aktion gewählt, die anschliessend dem Gegner minimale Chancen zusprechen:

$$a^*(x) := argmin_{a \in A_x} J'(x'_a, \omega),$$

wobei x'_a die Antwort des Gegners ist.

Diese Strategie wird, wie bereits schon erwähnt, in Tesauros TD-Gammon verwendet, einem sehr erfolgreichen Backgammonprogramm, dass Stärker als der Weltmeister spielt. Im Schach hingegen ist diese Vorgehensweise nicht sehr vorteilhaft, da doch jede Stellung sehr viele taktische Ereignisse beinhalten kann

und diese nur durch einen Suchbaum identifiziert werden können. Die Idee ist, nicht auf der Wurzel des Suchbaumes zu arbeiten, sondern das best-forcierte Blatt als Bewertungsgrundlage zu nehmen. Betrachten wir nun den MinMax-Algorithmus. Sei $J'_d(x,\omega)$ der erwartete Evaluationswert zur Stellung x, der durch $J'(\bullet,\omega)$ berechnet (in einer Tiefe d von x) aus erreichbar ist. Das Ziel ist es nun einen Parameter-Vektor ω zu finden, sodass $J'(\bullet,\omega)$ eine gute Näherung zum Optimum $J*$ist. Dazu wird für jede Stellung in einer Partie $x_1, x_2, ..., x_N$ eine temporale Differenz definiert

$$d_t := J'_d(x_{t+1},\omega) - J'_d(x_t,\omega)$$

und nun noch das aktualisieren des Vektors ω

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J'_d(x_t,\omega) \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right].$$

Sollte der Fall auftreten (im Schach nicht selten), dass es mehr als einen besten Zug gibt, so wählen wir einen zufälligen.

Basierend auf dem TD(λ)-Algorithmus wurde der TD-Leaf(λ)-Algorithmus [BaxTriWea2] entworfen, der nun anstatt der Wurzelknoten des Spielverlaufs $x_1, ..., x_N$ auf den Blättern arbeitet, die beim Alpha-Beta-PV-Algorithmus entdeckt wurden.

TDLeaf(λ)-Algorithmus

Der TDLeaf(λ)-Algorithmus [BaxTriWea2] arbeitet nun wie folgt:

Sei $J(\bullet,w)$ eine Klasse von Evaluierungsfunktionen, parametrisiert mit $\omega \in \mathbb{R}^k$. Seien weiterhin $x_1,...,x_N$ N Stellungen, die während einer Partie betrachtet werden und $r(x_N)$ das Resultat. Als Konvention gelte: $J(x_N,\omega) := r(x_N)$.

1. Für jeden Zustand x_i, berechne $J_d(x_i,\omega)$ unter Anwendung der MinMax-Suche bis zur Tiefe d von x_i unter Verwendung von $J(\bullet,w)$, welches die Blätter evaluiert. Beachte, dass d von Stellung zu Stellung variieren kann.
2. Sei x_i^l das Blatt, welches ausgehend vom Knoten x_i durch principal variation erreicht wird. Sollte es mehrere PV-Knoten geben, so wähle einen per Zufall. Beachte:

$$J_d(x_i,\omega) = J(x_i^l,\omega)$$

3. For $t = 1$ to $N-1$ berechne die Temporale Differenzen:

$$d_t := J(x_{t+1}^l,\omega) - J(x_t^l,\omega)$$

4. Update ω anhand der TD-Leaf(λ)-Formel:

$$\omega := \omega + \alpha \sum_{t=1}^{N-1} \nabla J(x_t^l,\omega) * \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_t \right]$$

3.3 Bilanz und Diskussion

Das Programm wurde mit einer Anfangsstärke von 1650 auf einem Internet Chess Server spielen gelassen. Nach 308 Spielen, in nur 3 Tagen, steigerte sich das Programm auf eine Stärke von 2150. Nach dem Einbau eines Eröffnungsbuches und weiteren kleinen Schachprogrammierraffinessen, hatte das Programm sogar eine Stärke von über 2500, was dem Grossmeisterniveau entspricht. Es stellt sich die Frage, inwieweit das Programm verbessert werden kann. Eine Möglichkeit wäre z.B. die Einführung einer Stellungsklassifizierung [Block]. Dadurch könnten typische Stellungsmuster identifiziert und durch individuell eingestellte Bewertungsparameter evaluiert werden. Der Stellungsklassifikator könnte eine grosse Partiensammlung

von Grossmeistern nutzen, deren Partien Zug für Zug untersuchen und als Vektor repräsentieren. Der aufgespannte *n*-dimensionale Raum kann nun durch beispielsweise **Expectation-Maximisation(EM)** aufgeteilt werden. Für die grossen Stellungsklassen werden Vektoren als Repräsentanten gespeichert und so kann nun vor der Stellungsbewertung geprüft werden, welchem Stellungstyp die Position am nächsten liegt. Als Resultat besitzt das Programm eine viel grössere Parameteranzahl und kann die Stellungen viel „feiner" behandeln. Das Problem würde nun darin liegen, dass bei einer so grossen Anzahl von Bewertungsfaktoren, die Lernrate α sehr viel langsamer konvergieren muss und es zeitlich dann wohl besser wäre das Programm gegen andere Schachmotoren spielen zu lassen. Zusätzlich muss dafür Sorge getragen werden, dass jeder Vektor die gleiche „Aufmerksamkeit" erfährt, also nicht einige übertrainiert und andere völlig vernachlässigt werden (das könnte bei Stellungstypen passieren, die einfach zu selten in einer Partie auftreten ...). Gelöst werden könnte dies durch mehrere Lernraten (für jeden Vector einen). Eine Anpassung des TD-Leaf(λ)-Algorithmus, speziell in der update-Funktion (nachdem der Algorithmus die Gradienten berechnet hat und nun den zulernenden Vektor anpassen möchte), wird nötig sein.

Baxter, Tridgell und Weaver geben an, dass das Spiel „gegen sich selbst" sehr viel langsamer und schlechter konvergiert, so hat z.B. eine KnightCap-Version, die in 600 Spielen gegen sich selbst gespielt hat, im Vergleichskampf gegen eine weitere Version, die 100 Spiele gegen Menschen gespielt hat, nur 11% der Spiele gewonnen [BaxTriWea4]. Das hat sicherlich auch damit zu tun, dass der Spielerpool auf dem Internetserver wesentlich reicher ist und ein Schachprogramm ohne zufällige Zugwahl-Parameter sich doch sehr deterministisch verhält (und damit wenig dazu lernt).

Ein weiterer Ansatz könnte sein, dass das Programm noch in Abhängigkeit der Stellung auf die Berechnung einiger Bewertungsfaktoren verzichtet, da die Berechnung beispielsweise teuer aber für die Evaluation nicht relevant sind.

Im Open-Source-Projekt **FUSC#** des Informatikfachbereichs der Freien Universität Berlin, ist für Ende 2003, Anfang 2004 die Implementierung dieser Idee geplant. Dann wird man sehen, inwieweit eine Stellungsklassifizierung realisierbar und die damit verbundenen Aussichten der Verbesserung der Reinforcement Learning-Erfolge in der Schachprogrammierung gerechtfertig sind.

3.4 Beispielpartie: KnightCap vs. KnightCap

Hier der Ausschnitt einer Beispielpartie von KnightCap gegen sich selber [BaxTriWea2]. Alle Evaluierungsparameter sind auf 0 gesetzt, bis auf die Materialkoeffizienten. Die ungefähre Spielstärke von KnightCap ist 1650, die auf dem SpielServer FICS ermittelt wurde, diese lässt sich aber nicht mit der Spielstärke von Menschen vergleichen, was die erstes 20 Züge einer Partie aufzeigen sollen.

1.*Sa*3 *Sc*6 2.*Sb*5 *Tb*8 3.*Sc*3 *Ta*8 4.*Tb*1 *d*6 5.*Ta*1 *Sa*5 6.*Sf*3 *Ld*7 7.*d*3 *Le*6 8.*Dd*2 *Sf*6 9.*a*3 *Sc*6 10.*De*3 *Sb*8

11.*Sd*4 *Lc*8 12.*Df*3 *Dd*7 13.*Sb*3 *Sc*6 14.*Kd*1 *Se*5 15.*Df*4 *Sg*6 16.*Dg*3 *c*5 17.*Se*4 *Sd*5 18.*Kd*2 *Db*5 19.*Sc*3 *Sxc*3

20.*bxc*3 *Da*6 ... und soweiter

3.5 Quellcodefragmente

Datei: td.c

Dieses Klasse zeigt den QuellCode-Abschnitt (Programmiersprache C), in dem der TD-Lambda-Algorithmus bei KnightCap angewendet wird.

```
#include "includes.h"
#include "knightcap.h"
#define TD_LAMBDA 0.7
#define TD_ALPHA (10/(EVAL_SCALE))
#define MAX_ROUNDS 4
#define MAX_SIZE 50
static int total_rounds;
```

```
struct max_struct {
   double val; int i,j,k;
};
static int max_compare(struct max_struct *m1, struct max_struct *m2) {
   if (m1->val > m2->val) return 1;
   else if (m1->val == m2->val) return 0;
   return -1;
}
extern struct state *state;
extern int player;
extern int dont_change[];
char *stage_name[] = {"OPENING", "MIDDLE", "ENDING", "MATING"};
#include "names.h"
static void p_coeff_vector(struct coefficient_name *cn, FILE *large, FILE *small) {
   int x;
   fprintf(large,"/* %s */\n", cn->name);
   if (small) fprintf(small,"/* %s */\n", cn->name);
   for (x=0; x<(cn+1)->index - cn->index; x++) {
      fprintf(large,"%7d,", coefficients[cn->index + x]);
      if (small) fprintf(small,"%7d,", coefficients[cn->index + x]/100);
   }
   fprintf(large,"\n"); if (small) fprintf(small,"\n");
}
static void p_coeff_array(struct coefficient_name *cn, FILE *large, FILE *small) {
   int x;
   fprintf(large,"/* %s */\n", cn->name);
   if (small) fprintf(small,"/* %s */\n", cn->name);
   for (x=0; x<(cn+1)->index - cn->index; x++) {
      fprintf(large,"%7d,", coefficients[cn->index + x]);
      if (small) fprintf(small,"%7d,", coefficients[cn->index + x]/100);
      if ((x+1)%10 == 0) {
         fprintf(large, "\n");
         if (small) fprintf(small, "\n");
      }
   }
   fprintf(large,"\n"); if (small) fprintf(small,"\n");
}
static void p_coeff_board(struct coefficient_name *cn, FILE *large, FILE *small) {
   int x, y;
   fprintf(large,"/* %s */\n", cn->name);
   if (small) fprintf(small,"/* %s */\n", cn->name);
   for (y=0; y<8; y++) {
      for (x=0; x<8; x++) {
         fprintf(large,"%7d,", coefficients[cn->index + x + y*8]);
         if (small) fprintf(small,"%7d,", coefficients[cn->index + x + y*8]/100);
      }
      fprintf(large,"\n"); if (small) fprintf(small,"\n");
   }
}
static void p_coeff_half_board(struct coefficient_name *cn, FILE *large, FILE *small) {
   int x, y;
   fprintf(large,"/* %s */\n", cn->name);
   if (small) fprintf(small,"/* %s */\n", cn->name);
   for (y=0; y<8; y++) {
      for (x=0; x<4; x++) {
         fprintf(large,"%7d,", coefficients[cn->index + x + y*4]);
         if (small) fprintf(small,"%7d,", coefficients[cn->index + x + y*4]/100);
      }
      fprintf(large,"\n"); if (small) fprintf(small,"\n");
   }
}
void dump_coeffs(char *fname, int round) {
   struct coefficient_name *cn;
   FILE *large, *small;
   int fd; int i;
   char fn[160];
   #if LARGE_ETYPE if (round >= 0) sprintf(fn, "/usr/local/chess/large_coeffs%d.h", round);      else sprintf(fn,"large_coeffs.h");
   large = (FILE *)fopen(fn, "w");
   sprintf(fn, "small_coeffs.h");
   small = (FILE *)fopen(fn, "w");
   #else if (round >= 0) sprintf(fn, "/usr/local/chess/small_coeffs%d.h", round);
   else sprintf(fn, "small_coeffs.h");
   large = (FILE *)fopen(fn, "w");
   small = NULL;
   #endif if (large == NULL) {
      perror(fname);
      return;
   }
   state->total_rounds = total_rounds;
   fprintf(large, "etype orig_coefficients[] = {\n");
```

```
   if (small) fprintf(small, "etype orig_coefficients[] = {\n");
   for (i=OPENING; i<=MATING; i++) {
      fprintf(large, "\n/* %%%s%% */\n", stage_name[i]);
      if (small) fprintf(small, "\n/* %%%s%% */\n", stage_name[i]);
      cn = &coefficient_names[0];
      coefficients = new_coefficients + i*__COEFFS_PER_STAGE__;
      while (cn->name) {
         int n = cn[1].index - cn[0].index;
         if (n == 1) {
            fprintf(large, "/* %s */ %d,\n", cn[0].name, coefficients[cn[0].index]);
            if (small) fprintf(small, "/* %s */ %d,\n", cn[0].name,
               coefficients[cn[0].index]/100);
         } else if (n == 64) {
            p_coeff_board(cn,large,small);
         } else if (n == 32) {
            p_coeff_half_board(cn,large,small);
         } else if (n % 10 == 0) {
            p_coeff_array(cn,large,small);
         } else {
            p_coeff_vector(cn,large,small);
         } cn++;
      }
   }
   fprintf(large, "};\n");
   if (small) fprintf(small, "};\n");
   fclose(large);
   if (small) fclose(small);
   fd = open(fname, O_WRONLY | O_CREAT | O_TRUNC, 0666);
   if (fd == -1) {
      perror(fname);
      return;
   }
   Write(fd, (char *)new_coefficients, __TOTAL_COEFFS__*sizeof(new_coefficients[0]));
  close(fd);
  return;
}
int td_dump(char *fname) {
   int i;
   etype sum;
    dump_coeffs(fname, total_rounds);
   sum = 0.0;
   for (i=0; i<__TOTAL_COEFFS__; i++) {
      sum += ABS(new_coefficients[i] - orig_coefficients[i]);
   }
   cprintf(0,"%d\n", sum);
   return 1;
}

/* routines for updating the evaluation function according to the method of temporal
differences */
#if LEARN_EVAL
int td_store_pos(Position *b) {
   state->leaf_pos[state->stored_move_num] = *b;
   print_board(state->leaf_pos[state->stored_move_num].board);
   ++state->stored_move_num;
   if (state->computer != 0) state->td_comp = state->computer;
   return 1;
}
/* calculate the partial derivative of the eval function with respect to each of
   the coefficients. computed numerically */
int td_gradient(float *big_grad) {
   etype v, v2;
   int i, n, m;
   etype delta = 100;
   float *grad;
   Position *b, b1;
   #if TEST_GRADIENT float error;
   etype v3, v4;
   #endif
   n = __COEFFS_PER_STAGE__;
   for (m = 0; m < state->stored_move_num; m++) {
      b = state->leaf_pos+m;
      lprintf(0, "%d ", m);
      /* sanity check */
      if (b->stage < OPENING || b->stage > MATING) {
         lprintf(0, "**Wrong stage in gradient calc: %d\n", b->stage);
         return 0;
      }
      b->flags &= ~FLAG_EVAL_DONE;
      b->flags &= ~FLAG_DONE_TACTICS;
```

```
      b1 = (*b);
      v = eval_etype(&b1, INFINITY, MAX_DEPTH);
      lprintf(0, "%d %d\n", v, b1.stage);
      state->leaf_eval[m].v = next_to_play(b)*v;
      if (!state->demo_mode) {
         state->leaf_eval[m].v *= state->td_comp;
      }
      coefficients = new_coefficients + b->stage*__COEFFS_PER_STAGE__;
      grad = big_grad + __TOTAL_COEFFS__*m + b->stage*__COEFFS_PER_STAGE__;
      for (i=0;i<n;i++) {
         b1 = (*b);
         v = eval_etype(&b1, INFINITY, MAX_DEPTH);
         coefficients[i] += delta;

         /* material only affects the eval indirectly via the board, so update the board */
         b1 = (*b);
         if (i > IPIECE_VALUES && i < IPIECE_VALUES+KING) create_pboard(&b1);
         v2 = eval_etype(&b1, INFINITY, MAX_DEPTH);
         grad[i] = next_to_play(&b1)*(v2 - v) / (float)delta;
         if (!state->demo_mode) {
            grad[i] *= state->td_comp;
         }
         #if TEST_GRADIENT coefficients[i] += delta;
         b1 = (*b);
         if (i > IPIECE_VALUES && i < IPIECE_VALUES+KING) create_pboard(&b1);
         v3 = eval_etype(&b1, INFINITY, MAX_DEPTH);
         coefficients[i] -= 2*delta;
         b1 = (*b);
         if (i > IPIECE_VALUES && i < IPIECE_VALUES+KING) create_pboard(&b1);
         v4 = eval_etype(&b1, INFINITY, MAX_DEPTH);
         error = next_to_play(&b1)*(v3 - v);
         if (!state->demo_mode) error *= state->td_comp;
         error -= 2*delta*grad[i];
         error /= delta;
         if (ABS(error)>0.05) {
            lprintf(0,"***coeff: %d grad: %f error: %f %e %e %e %e\n", i, grad[i], error, v,
                    v2, v3, v4);
         }
         #else coefficients[i] -= delta;
         #endif
      }
   }
   return n;
}
void td_save_bad(int fd, Position *b1) {
   int x;
   lseek(fd, 0, SEEK_END);
   if ((x = Write(fd, (char *)b1, sizeof(Position))) != sizeof(Position)) {
      lprintf(0,"***Error saving bad eval position %d %d\n", sizeof(Position), x);
   }
}
/* Updates the coefficients according to the TD(lambda) algorithm. */
int td_update() {
   int fd;
   int i,j,n,t;
   int argmax;
   int num_moves;
   int rounds = 0;
   float grad[300*__TOTAL_COEFFS__];
   double c, max;
   double dw[__TOTAL_COEFFS__];
   double olddw[__TOTAL_COEFFS__];
   double tanhv[MAX_GAME_MOVES];
   double d[MAX_GAME_MOVES];
   double oldnorm, newnorm, dotprod, angle;
   FILE *f;
   if (state->analysed) return 0;
   if ((f = (FILE *)fopen("rounds.dat", "r")) != NULL) {
      fscanf(f, "%d\n", &rounds);
      fclose(f);
   }
   if ((f = (FILE *)fopen("total_rounds.dat", "r")) != NULL) {
      fscanf(f, "%d\n", &total_rounds);
      fclose(f);
   }
   memset(dw, 0, __TOTAL_COEFFS__*sizeof(dw[0]));
   memset(olddw, 0, __TOTAL_COEFFS__*sizeof(dw[0]));
   #if DUMPING_TD_UPDATES fd = open("update.dat", O_RDONLY);
   if (fd != -1) {
      if (read(fd, olddw, __TOTAL_COEFFS__*sizeof(olddw[0])) !=
```

```
       __TOTAL_COEFFS__*sizeof(olddw[0])) {
      lprintf(0, "update file corrupt\n");
   } else {
      memcpy(dw, olddw, __TOTAL_COEFFS__*sizeof(olddw[0]));
   }
}
close(fd);
#endif
if (state->stored_move_num == 0 || state->stored_move_num > 300) {
   lprintf(0, "no gradient information: %d\n", state->stored_move_num);
   return 0;
}
memset(grad, 0, 300*__TOTAL_COEFFS__*sizeof(grad[0]));
if (state->ics_robot && result() == TIME_FORFEIT) num_moves = state->stored_move_num-1;
else num_moves = state->stored_move_num;
lprintf(0,"***moves: %d\n", num_moves);
n = __TOTAL_COEFFS__;
if (td_gradient(grad)) {
   lprintf(0,"gradients calculated\n");
} else {
   lprintf(0,"gradient error\n");
   return 0;
}
/* Squash the evals and compute the temporal differences */

tanhv[0] = tanh(EVAL_SCALE*state->leaf_eval[0].v);
for (t=0; t<num_moves-1; t++) {
   tanhv[t+1] = tanh(EVAL_SCALE*state->leaf_eval[t+1].v);
   d[t] = tanhv[t+1] - tanhv[t];
   if (state->predicted_move[t+1] == -1 && !state->demo_mode && state->rating_change < 0)
       d[t] = RAMP(d[t]);
}
/* work out the outcome */

if (state->demo_mode) {
   switch (state->won) {
      case STALEMATE: {
         if (NO_STALEMATE_LEARN) return 0;
         d[num_moves-1] = tanh(EVAL_SCALE*DRAW_VALUE) - tanhv[num_moves-1];
         break;
      }
      case 1: {
         d[num_moves-1] = 1.0 - tanhv[num_moves-1];
         break;
      }
      case 0: {
         d[num_moves-1] = -1.0 - tanhv[num_moves-1];
         break;
      }
   }
} else {
   switch (result()) {
      case STALEMATE: {
         if (NO_STALEMATE_LEARN) return 0;
         d[num_moves-1] = tanh(EVAL_SCALE*DRAW_VALUE) - tanhv[num_moves-1];
         break;
      }
      case 1: {
         d[num_moves-1] = 1.0 - tanhv[num_moves-1];
         break;
      }
      case 0: {
         d[num_moves-1] = -1.0 - tanhv[num_moves-1];
         break;
      }

      /* for time forfeited or resigned games we just assume the final eval was correct
      */
      case TIME_FORFEIT: {
         d[num_moves-1] = 0.0;
         break;
      }
   }
}
if (state->predicted_move[num_moves] == -1 && !state->demo_mode && state->rating_change
    < 0) d[num_moves-1] = RAMP(d[num_moves-1]);
lprintf(0,"outcome: %d %d %d\n", state->won, state->colour, state->position.winner);
for (i=0; i<num_moves; i++) {
   lprintf(0, "%d %d %lf\n", i, state->leaf_eval[i].v, d[i]);
}
```

```
   /* calculate the coefficient updates */
   max = 0.0;
   j=0;
   for (i=0; i<n; i++) {

      /* "FACTORS" are multiplicative and have disproportionally high derivatives so we
      don't adjust them */


      if (dont_change && i==dont_change[j]) {
         ++j;
         continue;
      }
      c = (1.0 - tanhv[0]*tanhv[0])*EVAL_SCALE*grad[i];
      for (t=0; t<num_moves; t++) {
         dw[i] += d[t]*c;
         if (t<num_moves-1) {
            c = TD_LAMBDA*c + (1-tanhv[t+1]*tanhv[t+1])* EVAL_SCALE*grad[(t+1)*n+i];
         }
      }
      if (ABS(dw[i]) > max) {
         max = ABS(dw[i]);
         argmax = i;
      }
   }
   lprintf(0,"max: %lf %d\n", TD_ALPHA*max, argmax);
   oldnorm = 0.0;
   newnorm = 0.0;
   dotprod = 0.0;
   for (i=0; i<n; i++) {
      oldnorm += ((double)new_coefficients[i]*(double)new_coefficients[i]);
      newnorm += (new_coefficients[i]+TD_ALPHA*dw[i])* (new_coefficients[i]+TD_ALPHA*dw[i]);
      dotprod += (new_coefficients[i] + TD_ALPHA*dw[i])*new_coefficients[i];
   }
   angle = 0.0;
   if (oldnorm != 0) angle = 180*acos(dotprod/sqrt(oldnorm*newnorm))/PI;
   lprintf(0, "change in angle: %lg\n", angle);
   f = (FILE *)fopen("angle.dat", "a");
   fprintf(f, "%g\n", angle);
   fclose(f);
   j = 0;
   for (i=0; i<n; i++) {
      if (dont_change && i==dont_change[j]) {
         ++j;
         continue;
      }
      if (rounds == MAX_ROUNDS) new_coefficients[i] += TD_ALPHA*dw[i];
   }
   #if DUMPING_TD_UPDATES fd = open("update.dat", O_WRONLY | O_CREAT | O_TRUNC, 0666);
   if (rounds == MAX_ROUNDS) {
      memset(dw, 0, __TOTAL_COEFFS__*sizeof(dw[0]));
      rounds = 0;
   }
   if (Write(fd, (char *)dw, __TOTAL_COEFFS__*sizeof(dw[0])) !=
   __TOTAL_COEFFS__*sizeof(dw[0])) {
      lprintf(0,"failed to write updates\n");
   }
   close(fd);
   ++rounds;
   f = (FILE *)fopen("rounds.dat", "w");
   fprintf(f, "%d\n", rounds);
   fclose(f);
   #endif lprintf(0,"updated coefficients\n");
   ++total_rounds;
   f = (FILE *)fopen("total_rounds.dat", "w");
   fprintf(f, "%d\n", total_rounds);
   fclose(f);
   state->analysed = 1;
   return 0;
}
#else void td_dummy(void) {}
#endif
```

Kapitel 4

Literatur

Schach & Schachprogrammierung

[Friedel] Steinwender, Friedel: *Schach am PC.* Markt&Technik 1995

[Heinz] Ernst A. Heinz: *Scalable Search in Computer Chess*. Vieweg 2000

[BarKraSch] Bartel, Kraas, Schrüfer: *Das grosse Computerschachbuch.* Data Becker 1985

[HaEd] Hamann, Eden: *Mit BASIC ran ans Schachprogramm.* DBV 1984

[PaKü] Pachman, Kühnmund: *Computerschach.* Heyne 1980

[Newborn] Newborn: *Computer Chess.* ACM 1975

[LeNew] Levy, Newborn: *How Computers play Chess*. Computer Science Press 1991

[Frick] Frickenschmidt: *SCHACH mit dem Computer*. Falken 1985

[Block] Block: *Die Geheimnisse der Schachprogrammierung.* Skript zum KI-Forschungprojekt FUSC# der FU-Berlin (in Vorbereitung)

Reinforcement Learning

[BaxTriWea1] J.Baxter, A.Tridgell, L.Weaver: *KnightCap: A chess program that learns by combining TD(λ) with game-tree search.* Publication 1999

[BaxTriWea2] J.Baxter, A.Tridgell, L.Weaver: *KnightCap: A chess program that learns by combining TD(λ) with minmax search.* Publication 1997

[BaxTriWea3] J.Baxter, A.Tridgell, L.Weaver: *TDLeaf(λ): Combining Temporal Difference Learning with Game-Tree Search.* Publication ?

[BaxTriWea4] J.Baxter, A.Tridgell, L.Weaver: *Learning to Play Chess Using Temporal Differences.* Machine Learning, 40, 243-263, 2000

[Wellner] Vorlesungsskript: Sommersemester 2002, TU Chemnitz

[Zhang] Zhang, Byoung-Tak: *Lernen durch Genetisch-Neuronale Evolution: Aktive Anpassung an unbekannte Umgebungen mit selbstentwicklenden parallelen Netzwerken.* Infix 1992

Künstliche Intelligenz

[Luger] Luger: *Künstliche Intelligenz.* Pearson Studium 2001

[Rojas] Rojas: *Theorie der neuronalen Netze.* Springer 1996