Das Schachprojekt FUSc#

Marco Block, André Rauschenbach, Johannes Buchner, Frank Jeschke, Raúl Rojas
[block|rauschen|buchner|jeschke|rojas]@inf.fu-berlin.de

November 2005

Zusammenfassung

Die AG Schachprogrammierung der Freien Universität Berlin wurde im Oktober 2002 gegründet und forscht seitdem auf dem Gebiet der Schachprogrammierung und beschäftigt sich mit der Entwicklung des Schachmotors *FUSc#*. Im Laufe der Zeit wurden verschiedene Brettdarstellungen und Algorithmen getestet. Mit einer nichtrekursiven Alpha-Beta-Variante in Matrixform konnte eine vielversprechende Leistungssteigerung erreicht werden, dem gegenüber stand aber ein großer Grad an Codekomplexität.

Als wichtigstes Forschungsziel bei dem seit Februar 2004 neu entwickelten Schachmotor *DarkFUSc#* gilt die Verwendung von Lernalgorithmen. Durch die Kombination von TD-Leaf(λ) und einer komplexen Stellungsklassifikation wurde ein neues Lernverfahren entwickelt. Nach den gespielten Partien optimiert *DarkFUSc#* die mehr als 56.000 Koeffizienten seiner komplexen Bewertungsfunktion eigenständig und verbessert so sein Stellungsspiel von Partie zu Partie. Als Trainings- und Testplattform für die verschiedenen FUSc#-Varianten (unter anderem existiert inzwischen auch eine Linux-Version) wurde der Fusch-OnlineSchachServer bereitgestellt, auf dem Maschinen und Menschen gegeneinander spielen und voneinander lernen können.

1 Arbeiten im Fuschprojekt

Zunächst einmal gilt der Dank allen freiwilligen Mitgliedern des FUSc#-Teams. In den vergangenen Jahren des Projektes haben viele am Code, dem Framework und den zahlreichen Auftritten und Veranstaltungen (z.B. Lange Nacht der Wissenschaften 2003 - 2005) mitwirken können. Auf diesem Wege, ein Dankeschön an: Christian Düster, Till Zoppke, Sebastian Dill, Maro Bader, Dominic Freyberg, Andreas Gropp, Falko Krause, Christian Ehrlich, Nima Keshvari, Ben-Fillippo Krippendorff, Michael Schreiber, Artem Petakov, Mike Krüger, Niklaas Görsch, Dennis Epple, Miguel Domingo, Jan Kretzschmar und Prof. Eberhard Letzner.

1.1 Projekthistorie

Die *AG Schachprogrammierung* entstand aus einem seminarähnlichen Treffen im Oktober 2002. Die Idee zu diesem Seminar, in dem Aspekte der Schachprogrammierung diskutiert und beleuchtet werden sollten, hatten Till Zoppke und Marco Block. Als Entwicklungssprache für das neue Open-Source-Projekt wurde C# gewählt. So wurde aus der ***FU Sch**achprogrammier AG* in Anlehnung an die Programmiersprache der Name *FUSc#* geboren.

Der Schachmotor wurde in ein UCI-Framework[1] gebettet, damit er mit anderen Schachprogrammen und menschlichen Spielern standardisiert kommunizieren kann. Als Alternative stand WinBoard[2] zur Verfügung, aber die Möglichkeiten unter UCI sind vielversprechender und die Entwicklung geht stetig voran [WinBoard-Webseite]. Der Grund für die Verwendung eines Standard-Protokolltyps bestand darin, dass die Entwicklung eines Schachmotors im Vordergrund stehen sollte und nicht der Entwurf einer neuen Oberfläche. Mit *Arena* (siehe Abschnitt 1.2.6) beispielsweise als nicht kommerzieller oder der *Fritz-Oberfläche* als kommerzieller Oberfläche standen bereits sehr gute Lösungen zur Verfügung. Das Schachprogramm *FUSc#* nahm an einigen Turnieren[3] [UCI-Webseite] teil und spielte zeitweilig im Internet z.B. auf dem deutschen Schach-Server [Chessbase-Webseite]. Grössere Erfolge liessen aber noch etwas auf sich warten.

Im Februar 2004 wurde zur Umsetzung einer Lernstrategie (Temporale Differenz) ein neuer, schnellerer Schachmotor entwickelt, der auf Teilen von *FUSc#* basierte und den Namen *DarkFUSc#* bekam. Dieser neue Motor besitzt eine vielversprechendere Brettdarstellung (Rotated Bitboards, siehe dazu Abschnitt 2.2) und kann, wie es der Name vermuten lässt, legale Züge anhand von Bitwortrepräsentationen generieren. Im Rahmen einer Diplomarbeit [Block 2004] wurde *DarkFUSc#* dahingehend verändert, dass er in der Lage ist, Änderungen an den mehr als 56.000 Koeffizienten seiner Bewertungsfunktion eigenständig nach einer gespielten Partie vorzunehmen. Als Test- und Trainingsumgebung wurde der deutsche Schach-Server gewählt. *DarkFUSc#* zeigte im Laufe des Trainings eine enorme Leistungssteigerung, auf die noch näher in Abschnitt 5.1 eingegangen wird.

Um den verschiedenen *FUSc#*-Versionen eine Spiel- und Trainingsplattform zu bieten, brachte Frank Jeschke im August 2004 einen neuen Schachspielserver (siehe Abschnitt 6) basierend auf dem Open-Source-Projekt *Lasker* zum laufen. Der Fusch-OnlineSchachServer [FUSch-Webseite] bietet Schachmaschinen und Menschen die Möglichkeit gegeneinander zu spielen und voneinander zu lernen. Es werden Turniere und besondere schachliche Ereignisse ausgetragen, um den aktuellen Stand der Entwicklung im Wettkampf zu erproben und ein Forum für Schachinteressierte und Schachprogrammentwickler zu bieten.

In der Zwischenzeit hat das FUSc#-Team in Person von André Rauschenbach mit dem neuen Java-Client eine komfortable und schnelle Zugangsmöglichkeit für Besucher der [FUSch-Webseite] entwickelt. Mit einem schnell erstellten Benutzerlogin kann jeder auf den Fusch-OnlineSchachServer gelangen und dort spielen oder den anderen beim Schachspielen zuschauen.

Nach Abschluss der Diplomarbeit wurde auf dem Fusch-OnlineSchachServer der **FIDE-Meister Ilja Brener (Elo 2375)** zu einem Schaukampf gegen den gut trainierten *DarkFUSc#* eingeladen. Das Ergebnis der zwei ausgespielten Partien endete

[1]Protokolltyp zwischen Schachprogramm und GUI, entwickelt von Stefan Meyer-Kahlen, dem Entwickler vom mehrfachen Weltmeisterschachprogramm *Shredder*. UCI=Universal Chess Interface.

[2]Tim Mann spezifizierte den Protkolltyp WinBoard und bietet eine gleichnamige Spieloberfläche an.

[3]Turniere und verschiedene Schachmotor-Ligen werden professionell organisiert von Alex Schmidt.

FM Ilja Brener $\mathbf{1,5} - \mathbf{0,5}$ DarkFUSc#

zugunsten des FIDE-Meisters (Remispartie siehe Abschnitt 8). Trotzdem war es für das FUSc#-Team ein Grund zum feiern, denn *DarkFUSc#* hatte am 15.09.2004 den ersten menschlichen Titelträger zum wackeln gebracht. Gestärkt durch dieses Erlebnis geht die Entwicklung weiter.

Die Algorithmen wurden im Rahmen einer Bachelorarbeit auf Herz und Nieren geprüft und weiter optimiert [Buchner 2005]. Nebenbei konnte die Performance durch die Umstellung auf die neue 64-Bit-Rechnergeneration verdoppelt werden.

1.2 Arbeits- und Testumgebung

1.2.1 Visual Studio .Net

Im Laufe des Projekts wurden verschiedene Entwicklungsumgebungen getestet, so zum Beispiel das Open-Source-Tool *Sharpdevelop* [SharpDevelop-Webseite] von Mike Krüger. Momentan wird der FUSc#-Code aber unter dem *Visual Studio .Net* (VS) von MicroSoft entwickelt. Der Grund für die Entscheidung pro VS war die grosse Erfahrung mit anderen Projekten, die einige FUSc#-Entwickler bereits hatten und der professionelle Hilfe-Support, den MicroSoft anbietet.

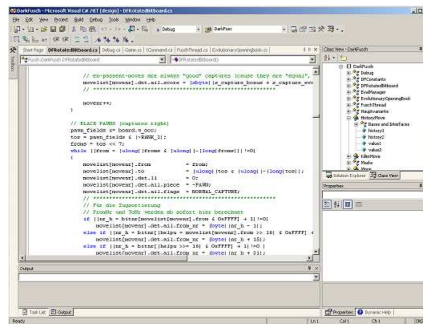

Abbildung 1: Entwicklung von FUSc# unter Visual Studio .Net

1.2.2 Proofing Tool *nprof*

Das Open-Source-Programm *nprof* [NProf-Webseite] ist ein Profiler für C#-Programme. Ein Profiler startet einen Prozess und liefert unter anderem Informationen über Anzahl und Dauer von internen Funktionsaufrufen. So lassen sich Optimierungen hinsichtlich der Befehlsreihenfolge oder der Wahl eines speziellen Problemlösungsansatzes im Programm aufspüren.

1.2.3 Zuggeneratorvalidierung mittels *perft*-Methode

Die *perft*-Methode liefert für eine konkrete Position bei einer festen Tiefe, die Anzahl der zu besuchenden Knoten im MinMax-Baum und dient so der Validierung des Zuggenerators (siehe dazu Tabelle 1) [Buchner Sem]. Beispielsweise wurde diese Methode im Schachmotor *Crafty* [Hyatt-Webseite] implementiert. Peter McKenzie bietet eine Sammlung ausgewählter und bereits mehrfach geprüfter Teststellungen an [McKenzie-Webseite].

1.2.4 FUSc# goes Linux

Sebastian Dill hat im Mai 2005 beim Einstieg in das FUSc#-Projekt damit begonnen, den Programmcode unter Linux mittels *Mono*[4] [Mono-Webseite] lauffähig zu bekommen. Nachdem einige wenige windowss-

[4] Laufzeitumgebung von C# unter Linux.

Tiefe	Anzahl Knoten im MinMax-Baum
1	20
2	400
3	8.902
4	197.281
5	4.865.609
6	119.060.324
7	3.195.901.860
8	84.998.978.956
9	2.439.530.234.167
10	69.352.859.712.417

Tabelle 1: „perft“-Ausgabe für die Startstellung

pezifische Programmteile angepasst werden mussten, ist *FUSc#* nun auch als Linuxversion im Downloadbereich der [FUSch-Webseite] erhältlich.

1.2.5 CVS

Die Verwendung eines CVS-Systems erleichtert die Arbeit in einem Projekt mit mehr als einem Programmierer sehr. Das FUSc#-Team arbeitet standardmässig mit dem kostenlosen *TortoiseCVS* [Tortoise-Webseite].

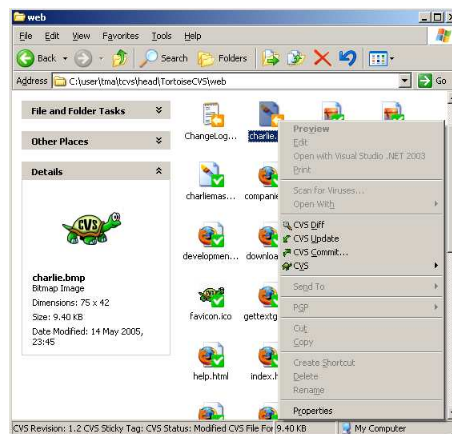

Abbildung 2: TortoiseCVS, Abbildung von *http://www.tortoisecvs.org/*

Verschiedene Code-Versionen von *FUSc#* können so leicht verwaltet werden und stehen in ihrer aktuellsten Version jederzeit zur Verfügung. Für die Einarbeitung in das Projekt steht ein ausführliches Tutorial [CVS-Tutorial] von André Rauschenbach zur Verfügung.

1.2.6 Arena

Die nichtkommerzielle Oberfläche *Arena* [Arena-Webseite] von Martin Blume bietet eine gute Tranzparenz bei der Fehlersuche und zur Optimierung von Schachprogrammen. Die Kommunikation und zahlreiche Bewertungsparameter während der Partie können sichtbar gemacht werden. Als Debugwerkzeug, als Test eines Schachprogramms unter Turnierbedingungen und als Client für zahlreiche Schachserver (der Fusch-OnlineSchachServer gehört dazu) ist das Programm sehr gut geeignet.

Abbildung 3: Arena, Abbildung von *http://www.playwitharena.com/*

2 Generation der Brettdarstellungen

2.1 Objectboard

Der Zuggenerator und die Brettdarstellung des ersten FUSc#-Schachmotors (Version 1.03) im März 2003 war rein objektorientiert. Dieser Ansatz hatte eine grösstmögliche Transparenz und war sehr fehlerunanfällig, da die Schnittstellen sehr genau beschrieben waren. Das Problem war die Performance. Der Kommunikationsaufwand zwischen den Klassen mit den zahlreichen Funktionen war sehr gross und der C#-Compiler war zu diesem Zeitpunkt nicht in der Lage, dieses Problem auf Maschinenebene zu lösen. So arbeitete der Zuggenerator zwar fehlerfrei, war aber extrem langsam. Aber schon zu diesem Zeitpunkt stellte sich heraus, dass die umfangreiche Bewertungsfunktion nicht allzu schlecht sein kann, denn trotz der geringen Suchtiefe (im Durchschnitt $3-4$) konnte *FUSc#* in Turnieren sehr gute Partien und Ergebnisse aufzeigen, so z.B. die Version 1.07 im Turnier zur Langen Nacht der Wissenschaften am Mathematik/Informatik-Fachbereich der Freien Universität Berlin (Abschlusstabelle 2, Abbildung 4).

Abbildung 4: *FUSc#* zur Langen Nacht der Wissenschaften 2003

Motiviert durch die Studienarbeit [Block 2003] und der Erkenntnis, dass nur ein taktisch stärkerer Schachmotor auch gute Lernerfolge bringen kann, mußte eine neue Brettdarstellung und ein neuer Zuggenerator implementiert werden. Die Wahl zugunsten der Rotated Bitboards und gegen das auch sehr oft verwendete 0x88-Brett [Moreland] wurde wegen der bevorstehenden 64-Bit-Rechnergeneration getroffen.

Im Februar 2004 entstand so ein neuer Schachmotor *DarkFUSc#*, der von Grund auf neu mit den Rotated Bitboards ausgestattet wurde und grosse Teile der Bewertungsfunktion, der Algorithmen sowie das komplette UCI-Framework von *FUSc#* übernahm.

Pos	Spielername	Wertung
1	Dauth, Benjamin	2290
2	Düster, Christian	2100
3	Domingo, Miguel	2038
4	Lane, Robin	-
5	Steffen, Rico	1971
6	Kuprat, Thomas	1975
7	Burghardt, Michael	1975
8	Martin, Mario	1900
9	Trösch, Thomas	2166
10	**FUSC# V1.07**	-
11	Kärcher, David	1368
12	Minski, Martin	2024
13	Rauch, Felix	1350
14	Schaller, Peter	1750
15	Wölter, Ulrich	1300
16	Remmo, Abdulrahim	1300

Tabelle 2: Abschlusstabelle - Lange Nacht der Wissenschaften 14.Juni 2003

Gespielt wurden 7 Runden im Schweizer System, mit 15 Minuten Bedenkzeit. Die Gegner von *FUSc#* bekamen zusätzlich 2 Minuten, da die Eingabe der Züge vom Brett in den Rechner eine kleine Verzögerung mit sich brachte. Sieger war der Elo-Favorit Benjamin Dauth, aber *FUSc#* konnte einen unerwarteten 10ten Platz erreichen, mit einer Glanzpartie gegen Martin Minski.

2.2 Rotated Bitboards

Die Verwendung der Bitboardtechnologie zahlt sich gerade mit der heutigen Rechnergeneration aus. Die Vorzüge dieser Darstellung sind die Verwendung der prozessorinternen, bitbasierten Befehle. So werden die Figurentypen (egal welcher Spielfarbe) in unterschiedlichen 64-Bitworten gespeichert (`board.pawns`, `board.knights`, `board.bishops`, `board.rooks`, `board.queens`, `board.kings`). Ein Bit steht für ein Feld und gibt an, ob an dieser Stelle diese Figur steht oder nicht.

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

Beispiel für die Startstellung: `board.knights`

Zusätzlich wird je ein Bitwort für die weissen und die schwarzen Figuren (`white_occupied`, `black_occupied`) gespeichert, so dass mit folgender, einfachen Operation die weissen Springer zu identifizieren sind:

```
white_knights  = board.knights & board.white_occupied;
```

Wurden noch mindestens 2 Operationen für diese Berechnung auf der 32−Bit Rechnergeneration benötigt, so ist es auf 64−Bit-Rechnern nur noch eine. Die Zuggenerierung erfolgt auch sehr einfach, beispielsweise können die Bauernzüge (ein Feld vor) durch eine einfache Shift-Operation generiert werden:

```
// liefert ein Bitwort mit den Zielfeldern der weissen Bauern
white_pawn_tos = (board.white_occupied & board.pawns) >> 8
```

$$
\left(
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}\\ \hline
\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}\\ \hline
\end{array}
\;\&\;
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
0&0&0&0&0&0&0&0\\ \hline
\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}\\ \hline
0&0&0&0&0&0&0&0\\ \hline
\end{array}
\right) >> 8 =
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}&\mathbf{1}\\ \hline
0&0&0&0&0&0&0&0\\ \hline
0&0&0&0&0&0&0&0\\ \hline
\end{array}
$$

Um nun individuelle Figurenmerkmale zu bewerten, stellt dich die Frage, wie es schnell möglich ist, eine einzelne Figur auf dem Brett zu finden (z.B. ein Springer) und diesen dann separat zu bearbeiten. Dazu gibt es zwei Funktionen[5], die eine wird mit `GetLeastSignificantBit()` (Get_LSB) bezeichnet und kann in einem Bitwort zunächst die letzte auftretende 1 identifizieren und so ein Bitwort bestehend aus Nullen und dieser 1 liefern. Die zweite Funktion nennt sich `ClearLeastSignificantBit()` (Clear_LSB) und löscht die letzte durch Get_LSB gefundene 1. So wird das elementeweise Durchlaufen in einer Schleife gespart.

Get_LSB:

bitwort & -bitwort ...liefert das Bitwort, mit der letzten gesetzten 1

Sei nun beispielsweise b=01010 mit dem Ziel, ein Bitwort zu erhalten, dass nur die letzte in b gesetzte 1 enthält. Um in der Zweierkomplementdarstellung ein Bitwort zu negieren, werden zunächst die einzelnen Bits gekippt und anschliessend zum Bitwort eine 1 addiert:

$$
\begin{aligned}
01010 \,\&\, -01010 &= 01010 \,\&\, (10101 + 1) \\
&= 01010 \,\&\, 10110 \\
&= 00010
\end{aligned}
$$

Clear_LSB:

bitwort = bitwort & (bitwort - 1) ...löscht die letzte 1 im Bitwort.

Als nächstes soll diese letzte 1 aus dem Bitwort entfernt werden, dazu:

$$
\begin{aligned}
01010 \,\&\, (01010 - 1) &= 01010 \,\&\, 01001 \\
&= 01000
\end{aligned}
$$

In einer Schleife können nun alle gesetzten Bits in einem Bitwort durchlaufen werden:

```
b = board.knights & board.white_occupied;
while (b = b&-b) {
      // Springer identifiziert ...
      b &= b-1;
}
```

Der Phantasie sind nun keine Grenzen gesetzt. Unterschiedliche Muster lassen sich in einem Bitmuster repräsentieren und entsprechend effizient verarbeiten.

Die Rotated Bitboards verwenden einen weiteren Trick für die Leistungssteigerung. Um beispielsweise die Läufer- und Turmzüge schneller zu generieren, existiert das Brettbitwort (gesamtes Brett), auf dem alle Figuren (schwarz und weiss) stehen, in vier Varianten:

[5] Ist möglich, da die Darstellung im Zweierkomplement vorliegt. Bei der Negation eines Bitwortes, werden alle Einträge invertiert und das erhaltene Bitwort anschliessend um 1 erhöht.

```
private int[] normal = {
        A1,B1,C1,D1,E1,F1,G1,H1,
        A2,B2,C2,D2,E2,F2,G2,H2,
        A3,B3,C2,D3,E3,F3,G3,H3,
        A4,B4,C3,D4,E4,F4,G4,H4,
        A5,B5,C4,D5,E5,F5,G5,H5,
        A6,B6,C5,D6,E6,F6,G6,H6,
        A7,B7,C6,D7,E7,F7,G7,H7,
        A8,B8,C7,D8,E8,F8,G8,H8
};
```

```
private int[] normal_to_l90 = {
        A1,A2,A3,A4,A5,A6,A7,A8,
        B1,B2,B3,B4,B5,B6,B7,B8,
        C1,C2,C3,C4,C5,C6,C7,C8,
        D1,D2,D3,D4,D5,D6,D7,D8,
        E1,E2,E3,E4,E5,E6,E7,E8,
        F1,F2,F3,F4,F5,F6,F7,F8,
        G1,G2,G3,G4,G5,G6,G7,G8,
        H1,H2,H3,H4,H5,H6,H7,H8,
```

```
private int[] normal_to_a1h8 = {
        A1,B2,C3,D4,E5,F6,G7,H8,
        A2,B3,C4,D5,E6,F7,G8,H1,
        A3,B4,C5,D6,E7,F8,G1,H2,
        A4,B5,C6,D7,E8,F1,G2,H3,
        A5,B6,C7,D8,E1,F2,G3,H4,
        A6,B7,C8,D1,E2,F3,G4,H5,
        A7,B8,C1,D2,E3,F4,G5,H6,
        A8,B1,C2,D3,E4,F5,G6,H7
};
```

```
private int[] normal_to_a8h1 = {
        A1,B8,C7,D6,E5,F4,G3,H2,
        A2,B1,C8,D7,E6,F5,G4,H3,
        A3,B2,C1,D8,E7,F6,G5,H4,
        A4,B3,C2,D1,E8,F7,G6,H5,
        A5,B4,C3,D2,E1,F8,G7,H6,
        A6,B5,C4,D3,E2,F1,G8,H7,
        A7,B6,C5,D4,E3,F2,G1,H8,
        A8,B7,C6,D5,E4,F3,G2,H1
```

Das Bitwort `normal` repräsentiert die „normale“ Brettstellung aller Figuren, wobei beim Lesen darauf zu achten ist, dass der erste Eintrag links, unten auf dem Brett zu finden ist und die Zeilen dementsprechend von oben nach unten verlaufen. Im Bitwort `normal_to_l90` steht das Wort um 90° nach links gedreht. Die beiden Bitworte `normal_to_a1h8` und `normal_to_a8h1` liefern die um 45° gedrehte Brettdarstellung.

Der Vorteil liegt nun bei der schnelleren Zuggenerierung. Bei der Initialisierung wurden alle möglichen Reihen- und Spaltenkonfigurationen ermittelt und die entsprechend möglichen Züge von jedem Feld aus berechnet und in einer Datenstruktur gespeichert. In der Suche müssen diese nun nicht generiert, sondern einfach ausgelesen werden. Rotated Bitboards sind nun desshalb nötig, um z.B. bei den Läufern die vorab berechneten Werte aus der Datenstruktur verwenden zu können, da die Bits der Diagonalen in der 45° gedrehten Brettdarstellung direkt aufeinanderfolgen. Sie können zusätzlich zum Startfeld als Index in die Datenstruktur genommen werden, um von einem Feld (Startfeld) alle möglichen Züge bei gegebener Diagonalkonfiguration zu erhalten. Analoges gilt für Türme und Damen.

Ein Nachteil ist allerdings, dass alle vier Bitworte bei Ausführung und Zurücknahme eines Zuges aktualisiert werden müssen.

In [Buchner Sem] befindet sich eine ausführliche Beschreibung der Bitboard-Implementierung von *DarkFUSc#*.

2.3 FUSc# goes 64-Bit

Die Umstellung des Zuggenerators bei *DarkFUSc#* auf die 64−Bit-Technologie brachte eine grosse Performancesteigerung [Buchner 2005]. Die Knotenanzahl pro Sekunde (nps) verdoppelte sich nahezu. Die nötigen Programmcodeänderungen waren aufgrund der Vorüberlegungen minimal.

3 Algorithmen

Der aktuelle FUSc#-Algorithmus ist ein iterativer Alpha-Beta-Algorithmus mit zahlreichen Heuristiken, wie z.B. die bekannten Killer-, Historyheuristik, futility pruning [Heinz 2000], extended futility pruning und speziellen Suchschranken (aspiration window [Plaat 1996]). Es werden verschiedene Transpositionstabellen [Dill Sem] und die Nullmovetechnik verwendet.

3.1 Nichtrekursiver Alpha-Beta-Algorithmus

Da C# in der aktuellen Version nicht die Möglichkeit einer Inline-Funktion[6] bietet, mussten zeitkritische Programmteile, die sehr oft ausgeführt werden, auch mehrfach im Code stehen, anstatt sie als Funktion zu implementieren. Die Performancesteigerung dadurch war beachtlich. Millionenfache Funktionsaufrufe sind sehr teuer, da viele Speicherumkopieroperationen nötig sein können.

Das brachte uns auf die Idee, das Rekursionsprinzip des Alpha-Beta-Algorithmus, das ausschliesslich auf Funktionsaufrufe ein und derselben Funktion beruht und möglicherweise mehrere Millionen mal aufgerufen wird, zu ändern und nach einer statischen Lösung zu suchen. Wir wählten eine Matrixdarstellung und simulierten die Reihenfolge der Zugfolgenabarbeitung in den entsprechenden Zuglisten in einer Matrix (siehe Abbildung 5).

tiefe \ zugliste	1	2	3	4	5	6	...
1	e2e4	d2d4	c2c4	f2f4	g1f3	b1c3	
2	e7e5	d7d5					
3	g1f3	d2d4	c2c4	b1c3	f2f4		
4	g8f6	d7d5					
5	f3e5	d2d3	d2d4	c1e2			
6	d7d6	f6e4	d7d5				
...							

Abbildung 5: Alpha-Beta-Variante in Matrixdarstellung

Es gab zunächst vier grundsätzliche Regeln:

Regel 1: Wenn möglich, setze einen Merkindex, gehe eine Tiefe weiter nach unten zum ersten Index.

Regel 2: Wenn es nicht weiter nach unten geht, dann bewerte die Stellung und gehe anschliessend (falls möglich) einen Index weiter in der Zugliste.

Regel 3: Wenn Regel 1 und 2 nicht komplett durchgeführt werden können, gehe eine Ebene zurück und dort bei dem Merkindex ein Element weiter.

Regel 4: Der Algorithmus terminiert bei Verwendung von Regel 3 in Tiefe 1.

Dem wirklich sehr vielversprechenden Performancegewinn stand ein sehr unübersichtlicher Programmcode mit viel Redundanz gegenüber, der nachträgliche Änderungen kaum bis garnicht zu lies. Eigentlich müsste dieser Optimierungsansatz nicht bei dem Programm, sondern beim Compiler verwendet werden. Aus Gründen der Weiterentwicklung wurde die rekursive Variante wieder eingeführt.

Im Unterschied zu der Vorversion des Algorithmus werden die Variablen global gehalten. Das spart neben den Umkopieroperationen auch noch Speicherplatz ein.

[6]Als Inline-Funktion wird eine Funktion bezeichnet, die beim Compilierungsvorgang an jede Aufrufposition in den Programmcode kopiert wird. Sie ersetzt den Aufruf mit dem Funktionskörper. Das lässt den Code übersichtlich und hat den Vorteil, „teure“ Funktionsaufrufe einzusparen.

3.2 Zeitmanagement

Das Problem vieler Schachprogramme in Partien mit geringer Bedenkzeit ist das Zeitmanagement. Zwei Ansätze, die in Amateurprogrammen oft verwendet werden, sind das Rechnen bis zu einer festen Tiefe und eine feste Zeitvorgabe für die Berechnung eines Zuges. Die feste Suchtiefe birgt die Gefahr, dass eine Stellung die sehr komplex ist und die Ruhesuche enorm ausreizt und belastet sehr viel Zeit in Anspruch nehmen kann und daher für zeitkritische Phasen ungeeignet ist. Nicht so problematisch, aber sehr ineffektiv ist die Vorgabe einer festen Zeit. Der iterative Suchalgorithmus hat beispielsweise die Tiefe x fast fertig berechnet und wird abgebrochen, dann steht nur noch das Ergebnis der vorhergehenden Suchtiefe $x-1$ zur Verfügung. Diese „verlorene“ Zeit lässt sich aber durch eine einfache Heuristik oft einsparen.

Zu diesem Zweck haben wir viele Testläufe mit den FUSc#-Algorithmen unternommen und heuristische Werte für die zu besuchenden Knotenanzahlen in den entsprechenden Suchtiefen ermittelt. Ziel war es den Faktor herauszufinden, der eine Abschätzung darüber liefert, inwieweit sich das Zeitverhältnis der entsprechenden Suchtiefen untereinander verhält.

Das FUSc#-Zeitmanagement arbeitet wie folgt. Zunächst legen wir beispielsweise in der Blitzphase ($2-10$ Minuten Bedenkzeit noch für den Rest der Partie übrig) ein 13-tel der restlichen Zeit als Bedenkzeit für die Suche fest. Dann wird eine Optimierung dahingehend vorgenommen, dass nach einem iterativen Schritt geprüft wird, ob die Zeit, die für die Tiefe x benötigt wurde, mal 10 genommen kleiner der restliche Zeit ist. Wenn ja, bearbeite die nächste Tiefe $x+1$, wenn nein liefere die Ergebnisse der Suchetiefe x zurück.

Die Zeitersparnis ist enorm und das Verhalten von *DarkFUSc#* in Blitzpartien oder Zeitnotphasen ausgezeichnet. Der nächste logische Schritt ist eine Vorschrift, die den Faktor 10 nicht fest sondern variabel gestaltet. Je nach Stellungstyp und Partiephase sollte sich der Faktor eigenständig, abhängig von den benötigten Zeiten, anpassen.

3.3 FUSc#-Algorithmus-Framework

```
public void ComputeMove() {
   Game.Manager.NewBoard.resetAllComputeData();
   Game.Manager.NewBoard.resetAllComputeHash();
   String aktBestNextMove = "";
   String lokalbestNextMove = "";
   // *****************************************************************
   // STELLUNGSKLASSIFIKATION
   evalindexClassifier = Klassifikator();
   aktBestMove.from = 0; // quasi moveinit
   // *************************************************
   // Eroeffnungsbuch
   if (Game.Manager.UsingOpeningBook)
      oBookMove = Game.Manager.EvoOpeningBook.ComputeMove();
   if ((oBookMove.from!=0)&&(Game.Manager.UsingOpeningBook)) {
      // Eroeffnungsbuchzug wird gesetzt
      aktBestMove=oBookMove;
      // da Zug gefunden wurde, gibt es auch einen Pfad
   }
   else {
      // *******************************************************************
      // Nun muessen wir ermitteln, in welcher Zeitphase wir sind, also
      // Bullet, Blitz oder Standard
      zeitmultiplikator = 10; // STANDARD 12
      if (TimeToMove<10*60*1000) zeitmultiplikator = 10; // BLITZ 13
      if (TimeToMove<2*60*1000)  zeitmultiplikator = 10; // BULLET 16
      // *******************************************************************
      // kein OBOOK-Move, also rechne selber ;)
      int alpha = -MATT;
      int beta = MATT;
      int merkealpha = 0;
      int merkebeta = 0;
      int iterativ = 0; //1;
      int suche = alpha;
      lokalbestMove = new Move();
      lokalbestPath = "";
      lokalbestScore = alpha;
      lokalBestDepth = 0;
      lokalBestMillis = 0;
      aktBestMove.from = 0;
      HauptVariante.Clear();
      newtimer.Start();
```

```
double milliseconds=0;
AlphaBeta_breaked = false;
int window = 50;
int windowSecond = 400;
int schritt = 0;
while(true) {
   milliseconds = newtimer.Stop();
   // *****************************************************************
   // Initialisierung:
   MAXTiefe = iterativ;
   MAXRuheTiefe = 0;
   iterativ+=1;
   schritt++;
   // *****************************************************************

   // *****************************************************************
   // Einhaltung der vorgegebenen Tiefe:
   if ((DepthToDo>0) && (iterativ-1>=DepthToDo)) break;
   // *****************************************************************

   // *****************************************************************
   // Einhaltung der vorgegebenen Zeit
   // NUR! wenn nicht unendlich, oder feste Tiefe
   if (!constTime) {
      milliseconds = newtimer.Stop();
      if ((AlphaBeta_breaked)||(milliseconds>TimeToMove)) break;
   }
   // *****************************************************************
   // *****************************************************************
   // Optimierung, falls wenig Zeit, gehe nicht in die nächste Tiefe!
   // (heuristisch ermittelt)
   if (TimeToMove < milliseconds*zeitmultiplikator) {
      Console.WriteLine("TimeToMove:"+TimeToMove+" milliseconds:"+milliseconds+"
                       zeitmultiplikator:"+zeitmultiplikator);
      break;
   }
   // *****************************************************************
   // ********************************************************************
   // der eigentliche Alpha-Beta-Aufruf:
   if (schritt==1) {
      initAlphaBeta(alpha, beta, wtm==1, 1, true);
      aktBestMove.det.ail.from_nr = 0;
      lokalbestPath = "";
      lokalbestScore = 0;
      suche = AlphaBeta();
      lokalbestMove = aktBestMove;
      lokalbestPath = HauptVariante.ToString();
      lokalbestScore = suche;
   } else {
      // *****************************************************************
      // Nun suchen wir mit dem "Aspiration-Verfahren" weiter und öffnen
      // Ein kleines Fenster um alpha herum...
      merkealpha = suche - window;
      merkebeta = suche + window;
      // *****************************************************************

      initAlphaBeta(suche-window, suche+window, wtm==1, iterativ, true);//Ruhesuche!
      suche = AlphaBeta();
      if (suche<=merkealpha) {
         Game.Manager.NewBoard.resetAllComputeHash();
         // *****************************************************************
         // Optimierung, falls wenig Zeit, gehe nicht in die nächste Tiefe!
         // (heuristisch ermittelt)
         if (!constTime) {
            milliseconds = newtimer.Stop();
            if (TimeToMove<milliseconds*zeitmultiplikator) break;
         }
         // *****************************************************************
         initAlphaBeta(suche-windowSecond, suche+1, wtm==1, iterativ, true);//Ruhesuche!
         suche = AlphaBeta();
         if (suche<=merkealpha-windowSecond) {
            // Hash-Werte sind falsch!! also löschen
            Game.Manager.NewBoard.resetAllComputeHash();

            // *****************************************************************
            // Optimierung, falls wenig Zeit, gehe nicht in die nächste Tiefe!
            // (heuristisch ermittelt)
            if (!constTime) {
               milliseconds = newtimer.Stop();
```

```
                    if (TimeToMove<milliseconds*zeitmultiplikator) break;
                 }
                 // ****************************************************************
                 initAlphaBeta(-MATT, merkealpha-windowSecond+1, wtm==1, iterativ, true);//Ruhesuche!
                 suche = AlphaBeta();
              }
           } else if (suche>=merkebeta) {
              // Hash-Werte sind falsch!! also löschen
              Game.Manager.NewBoard.resetAllComputeHash();
              // ****************************************************************
              // Optimierung, falls wenig Zeit, gehe nicht in die nächste Tiefe!
              // (heuristisch ermittelt)
              if (!constTime) {
                 milliseconds = newtimer.Stop();
                 if (TimeToMove<milliseconds*zeitmultiplikator) break;
              }
              // ****************************************************************

              initAlphaBeta(suche-1, suche+windowSecond, wtm==1, iterativ, true);//Ruhesuche!
              suche = AlphaBeta();
              if (suche>=merkebeta+windowSecond) {
                 // Hash-Werte sind falsch!! also löschen
                 Game.Manager.NewBoard.resetAllComputeHash();
                 // ****************************************************************
                 // Optimierung, falls wenig Zeit, gehe nicht in die nächste Tiefe!
                 // (heuristisch ermittelt)
                 if (!constTime) {
                    milliseconds = newtimer.Stop();
                    if (TimeToMove<milliseconds*zeitmultiplikator) break;
                 }
                 // ****************************************************************
                 initAlphaBeta(merkebeta+windowSecond-1, MATT, wtm==1, iterativ, true);//Ruhesuche!
                 suche = AlphaBeta();
              }
           }
        }
        // ********************************************************************

        ...
        // Ausgabe der berechneten Daten
        ...
     }
  }
```

3.4 Bubble-Heuristik

Im Alpha-Beta-Algorithmus werden bei den inneren Knoten des Suchbaumes die Zuglisten generiert. Die Züge in den Zuglisten besitzen verschiedene Werte (`movescore`), die den qualitativen Unterschied zwischen ihnen bei der Suche angeben sollen. Beispielsweise besitzt der Zug aus der Transpositionstabelle den grössten Wert, damit er nach einer Sortierung als erster abgearbeitet wird und möglicherweise so zu einem frühen Cut führen kann. Diese Sortierung der Züge lässt sich beispielsweise mit *BubbleSort* vornehmen. Da die zusortierende Liste relativ klein (im Schnitt |Zugliste|=45) und durch den Zuggenerator schon etwas vorsortiert ist arbeitet *BubbleSort* schneller als *QuickSort*. Nun hat sich aber in vielen Testläufen herausgestellt, dass nicht die komplette Zugliste sortiert werden muss, sondern es genügt die besten 3 Züge nach vorn zu sortieren. Diese kleine nützliche Heuristik haben wir in Anlehnung an den verwendeten Sortieralgorithmus *Bubble-Heuristik* genannt.

4 Bewertungsfunktion

Die Implementierung der Bewertungsfunktion ist stark von der Brettdarstellung abhängig. Das folgende Beispiel zeigt die Identifikation von Doppelbauern, isolierten Bauern und der Bewertung der Bauernposition, als Nebeneffekt liefert die Funktion Informationen über halboffene und offene Linien, die in späteren Abschnitten verwendet werden.

```
...
// ****************************************************************
// Initialisierung der Parameter:
```

```
e_bauern        = board.pawns & board.w_occ;
e_AllWhitePawns = e_bauern;
e_AllBlackPawns = board.pawns ^ e_bauern;
e_half_pawn_open_files = 0xFFFFFFFFFFFFFFFFUL;
// unter den eigenen Bauern isoliert
e_w_isolated = 0;
// aber ein gegnerischer Bauer versperrt das weiterlaufen
e_w_isolated_of = 0;
// aktuelle Position der zu behandelnden Figur
e_square = 0;
// ******************************************************************
while ((e_bauer =(ulong)(e_bauern & (ulong)(-(long)e_bauern)))!=0)
{
   if ((nr_h = bitnr[e_bauer & 0xFFFF] + 1)!=0)
      e_square = (byte)(nr_h - 1);
   else if ((nr_h = bitnr[(helpu = e_bauer >> 16) & 0xFFFF] + 1)!=0 )
      e_square = (byte)(nr_h + 15);
   else if ((nr_h = bitnr[(helpu >>= 16) & 0xFFFF] + 1)!=0 )
      e_square = (byte)(nr_h + 31);
   else if ((nr_h = bitnr[(helpu >>= 16) & 0xFFFF] + 1)!=0 )
      e_square = (byte)(nr_h + 47);
   // ******************************************************************
   // nun haben wir einen weissen Bauern identifiziert
   // die Position liefert: e_square
   // ******************************************************************
   // ******************************************************************
   // HALF-OPEN-FILE
   // Wir haben einen Bauern in dieser Spalte (file) gefunden, sie ist zu
   // diesem Zeitpunkt also nur noch "halboffen"
   // Streichen das file aus dem openfile-ulong
   e_pawn_file = file[e_square%8];
   e_half_pawn_open_files ^= e_pawn_file;
   // ******************************************************************
   // ******************************************************************
   // PAWN POSITION
   evalwert += evaluationLIST[evalindexClassifier, e_square+1578];
   // ******************************************************************
   // ******************************************************************
   // ISOLATED PAWN
   // nun wollen wir nach isolierten Bauern ausschau halten
   if ((e_mask_pawn_isolated[e_square] & e_AllWhitePawns)==0)
   {
      e_w_isolated++;
      //Console.WriteLine("e_isolated found on "+e_square);
      //DisplayBoard();
      // Steht noch ein Bauer vor diesem, dann ist er ein
      // "wirklicher" isolierter Bauer und demnach schlecht.
      // Es könnte sich bei diesem aber noch um einen
      // Freibauern oder Kandidaten handeln!
      if ((e_mask_over_figur[e_square] & e_AllBlackPawns)==0)
      {
         // kein Bauer davor ... also möglicherweise ein Freibauer!
         e_w_isolated_of++;
      }
      else
      {
         // er ist blockiert!
         //evalwert += e_blockedisolated;
         evalwert += evaluationLIST[evalindexClassifier, 1569];
      }
      // er ist isoliert, also bewerte ihn
      //evalwert += e_isolated[e_square%8];
      evalwert += evaluationLIST[evalindexClassifier, 7 + e_square%8];
   }
   // ******************************************************************
   // ******************************************************************
   // DOUBLE PAWN
   // befinden sich auf diesem file noch mehr eigene Bauern?
   e_zaehler=0;
   e_pawn_file &= e_AllWhitePawns;
   e_pawn_file ^= bit[e_square];
   while ((ulong)(e_pawn_file & (ulong)(-(long)e_pawn_file))!=0)
   {
      e_zaehler++;
      e_pawn_file&=e_pawn_file-1;
   }
   if (e_zaehler>0)
      evalwert += evaluationLIST[evalindexClassifier, 15 + e_square % 8];
   // ******************************************************************
   // der nächste Bauer ist an der Reihe
```

```
    e_bauern &= e_bauern-1;
}
...
```

Die Bewertungsfunktion unterscheidet pro Stellungsklasse 1.706 Koeffizienten. Für die 33 verschiedenen Stellungstypen stehen demnach 56.298 Gewichte zur Verfügung. Eine detailierte Besprechung des FUSc#-Codes und die Auswahlkriterien der 33 Stellungstypen befinden sich in [Block 2004].

5 Lernalgorithmen

Der Forschungsschwerpunkt liegt bei den Lernalgorithmen. Ideen und Konzepte für die Umsetzung der Kombination von Plänen im Schach und den Suchalgorithmen liegen bereits vor und werden in naher Zukunft umgesetzt. Bisher wurden zwei Ansätze verfolgt, zum einen die dynamische Anpassung der Eröffnungsbücher nach wahrscheinlichkeitstheoretischen Aspekten und zum anderen die Optimierung der Bewertungskoeffizienten durch eine für das Schachspiel angepasste Temporale-Differenz-Methode.

5.1 Temporale Differenz

Temporale Differenz(TD) wurde das erste mal von A.L.Samuel beschrieben [Samuel 1959] und später von R. Sutton formalisiert [Sutton 1988]. Der erste Einsatz von TD in der Schachprogrammierung wurde von M.Gherrity mit seinem Programm *SAL* [Gherrity 1993] unternommen und führte noch nicht zu einem Erfolg. Gründe dafür waren zum einen der mit *GNUChess [GNUChess]* zu stark gewählte Gegner und die daraus resultierende Konsequenz, dass jede Stellung scheinbar zur Niederlage führt. Zum anderen war aber auch die unzureichende Ausstattung des Programms, was die damals verwendeten Schachprogrammierstandards betrifft, ein wichtiger Grund für die schlechten Ergebnisse.

S.Thrun entwickelte das Schachprogramm *NeuroChess*, das ein Neuronales Netz als Bewertungsfunktion verwendet und eine TD-Methode, basierend auf den Wurzelknoten des Suchbaums, um die Koeffizienten der Bewertungsfunktion zu optimieren [Thrun]. Obwohl die Lern- und Spielergebnisse gegen *GNUChess* wesentlich besser waren, zweifelte Thrun daran, dass die Verwendung von TD-Methoden bei Schachprogrammen in der Zukunft zu einem Durchbruch verhelfen können.

Als nun aber 1997 der damalige Weltmeister Garry Kasparow gegen das von IBM in 5 Jahren entwickelte Schachprogramm *DeepBlue* verlor [King 1997] und bei den nachträglichen Analysen herauskam, dass bei der Optimierung der Koeffizienten (siehe Abbildung 6) die von G.Tesauro veröffentlichte neue TD-Methode [Tesauro] verwendet wurde, gab es wieder eine intensive Forschung auf diesem Gebiet.

Tesauro verwendete in seinem Backgammonprogramm *TDGammon* den TD(λ)-Algorithmus. Die Programmstärke nahm beim Lernen stark zu, so dass eine Version, die 1.500.000 Partien zuvor gespielt hatte, gegen Bill Robertie einen der stärksten Backgammonspieler der Welt in 40 Partien 39 mal gewann.

Der TD(λ)-Algorithmus wurde entsprechend der Unterschiede zwischen Backgammon und Schach für das Schachprogramm *KnightCap* von A.Tridgell, J.Baxter und L.Weaver weiterentwickelt und angepasst [BaxTriWea 1998]. Der Unterschied bestand darin, nicht auf den Wurzelknoten, sondern auf den best forcierten Blattknoten einer Stellung zu arbeiten. Das Programm *KnightCap* konnte seine Bewertungskoeffizienten mit dem Algorithmus TD-Leaf(λ) von Partie zu Partie verbessern und erreichte ein erstaunliches Niveau.

Durch diese Arbeit motiviert, wurde in *DarkFUSc#* mit TD-Leaf-ComplexEval(λ) (siehe Algorithmus 1) eine Kombination aus komplexer Stellungsklassifikation und TD-Leaf(λ) mit Erfolg entwickelt [Block 2004].

Das Problem bei einer sehr komplexen Stellungsklassifizierung besteht aber in der Dauer des Lernvorgangs. Einige Stellungstypen werden sehr selten erreicht, so dass es sein kann, dass *DarkFUSc#* den einen Stellungstyp schon sehr gut verstehen und spielen kann, aber einen anderen nicht oder kaum.

Durch das Spielen und Trainieren von *DarkFUSc#* auf dem Fusch-OnlineSchachServer (siehe dazu Abschnitt 6) wird aber eine stetige Verbesserung in den nächsten Monaten und Jahren erwartet.

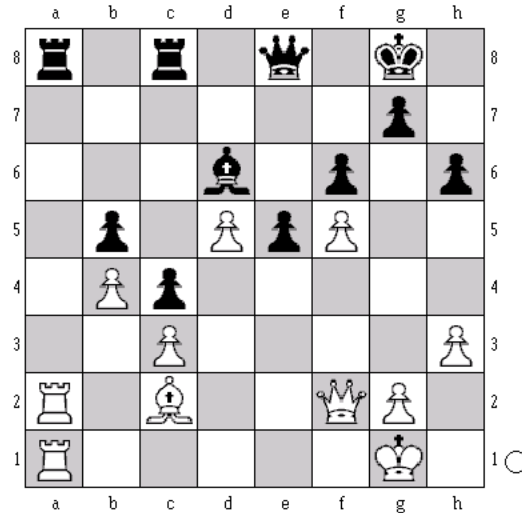

Abbildung 6: Deep Blue - Garry Kasparow, Spiel 2 beim Rematch 1997.

In dieser Stellung hätte Deep Blue mit den normalen Koeffizienten Db6 gespielt, was ein typischer Computerzug gewesen wäre, denn er hätte auf einen Bauerngewinn spielen können. Kasparow und das Publikum waren nun sehr überrascht, als Deep Blue Le4! spielte, ein sehr positioneller Zug, und damit jegliches Gegenspiel erstickte. Die Drohung Db6 ist nun sehr viel stärker. Kasparow verlor diese Partie schließlich und lobte das positionelle Verständnis des Programms besonders in dieser Stellung.

Algorithm 1 TD-Leaf-ComplexEval(λ)

Sei $J(\bullet, w)$ eine Klasse von Evaluierungsfunktionen, parametrisiert
mit $\omega \in \mathbb{R}^k$. Seien weiterhin $x_1, \ldots, x_N$ N eigene Stellungen, die während
einer Partie betrachtet werden und $r(x_N)$ das Resultat.
Als Konvention gelte: $J(x_N, \omega) := r(x_N)$.
Vorbereitung in einer Partie:
  $p(x_i)$, letzte Stellung der Hauptvariante (Länge p) von
  Stellung x_i mit $x_{i_1}, x_{i_2}, \ldots, x_{i_p}$
Lernschritt:
  1. for j=x_N to x_1 do begin
    1.1 gehe zu Stellung $p(x_j)$
    1.2 berechne den Gradienten von $J(p(x_j), \bullet)$
    1.3 berechne die Temporalen Differenzen
      $d_t := J(p(x_{t+1}), \omega) - J(p(x_t), \omega)$
      (mit Ausnahme von d_N)
    1.4. gehe zu Stellung x_j zurück
  end
  2. Update ω anhand der TD-Leaf(λ)-Formel:

$$\omega_k := \omega_k + \alpha_k \sum_{t=1}^{N-1} \nabla J(p(x_t), \omega_k) * \left[\sum_{j=t}^{N-1} \lambda^{j-t} d_j \right]$$

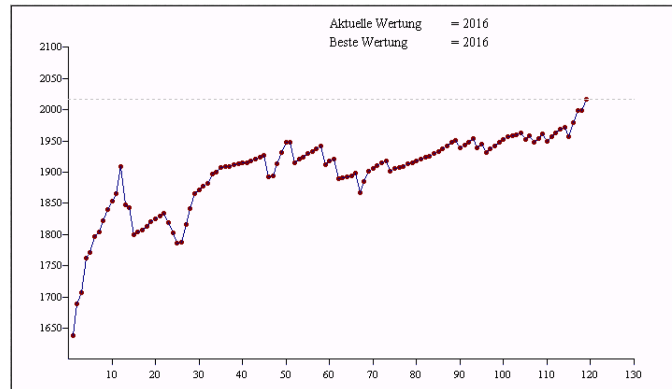

Abbildung 7: Bewertung auf dem deutschen Schachserver [Chessbase-Webseite] nach 119 Blitz-Partien

5.2 Evolutionäres Eröffnungsbuch

DarkFUSc# verwendet, wie schon *FUSc#* in der letzten Version 1.10 vor der Suche ein evolutionäres Eröffnungsbuch. Im Eröffnungsbuch werden Stellungen mit den entsprechenden Spielergebnissen gespeichert. Eine Stellung liefert ein Tupel (Sieg, Remis, Niederlage, Verlauf der letzten 20 Partien). In Abhängigkeit von der Aktualität der gespielten Partie wird ein Score für diese Stellung geliefert. Die Wahl zur vielversprechendsten Stellung wird mit einer gewissen Wahrscheinlichkeit ausgeführt.

6 Fusch-Schachspielserver

Für Trainings- und Testzwecke wurde der Fusch-OnlineSchachServer entwickelt [FUSch-Webseite]. Die bereits lauffähige in der Programmiersprache C geschriebene Open-Source Serverversion *Lasker* von der Universität Karlsruhe wurde installiert und im Laufe der Zeit umprogrammiert und den Erfordernissen angepasst.

Auf dem Server laufen permanent zwei Programme, die bestimmte Dienste leisten. Der Bot *Gambit* organisiert Tabellen, Listen und Turnierergebnisse für die Webseite und gibt dem Javaclient auf Anfrage wichtige Informationen über die Spieler und die gespielten Partien. Das Programm *Hal* verwaltet die Server-Turniere.

6.1 FUSc#-Webseite

Der Serverbot Gambit generiert alle für die Server-Webseite benötigten Tabellen und Listen. Diese Tabellen werden mit einem php-Skript eingelesen und fast zeitecht angezeigt. So können alle Informationen über eingeloggte Spieler und laufende Partien und Turniere, ohne den Server besuchen zu müssen, auf der Webseite verfolgt werden.

Alle Ergebnisse und die Abschlusstabellen beendeter Turniere werden automatisch für die Webseite aufbereitet. Durch die permanente Pflege, die Aktualität und die recht hohe Besucherzahl hat die Webseite bei der Anfrage „*Schachprogrammierung*“ auf der [Google-Webseite] den ersten Hit.

6.2 Javaclient

Für den Fusch-Schachspielserver wurde eine Javapplet geschrieben, um den Zugang zum Server komfortabler zu gestalten. Bisher wurden Arena 1.2.6 und WinBoard [WinBoard-Webseite] als Zugangssoftware empfohlen, aber die konsolenlastige Kommunikation schreckte viele Besucher der Webseite ab, unseren Server zu verwenden.

Abbildung 8: Fuschserver *http://www.fuschmotor.de.vu*

Der Javaclient ermöglicht eine leichte Bedienung. Es gibt eine Liste der aktuell auf dem Server eingeloggten Spieler mit vielen Informationen. Eine Chatmöglichkeit mit einem Spieler, einer Gruppe von Spielern oder allen bietet viel Raum für Kommunikation. Die Serveradministratoren sind farblich hervorgehoben. Sie beantworten jederzeit Fragen und helfen beim Umgang mit dem Programm.

Neben eine Spielfunktion, lässt sich auch anderen beim Spielen zuschauen. Der Server verwaltet ein Elo-Punktesystem, das von Arpad Elo in den sechziger Jahren entwickelt und auf dem FIDE-Kongress in Siegen 1970 eingeführt wurde. Momentan ist **FM Ilja Brener** mit 2502 Punkten der stärkste menschliche Spieler und **Deep Shredder** 9[7] mit 2482 Punkten der stärkste Computer auf dem Server.

7 Diskussion und Ausblick

Das FUSc#-Projekt ist ein studentisches Projekt, an dem nicht täglich geforscht und weiterentwickelt wird. Die Motivation zur Entwicklung eines Schachmotors war es mit Spass an der Sache eigene Ideen und Vorstellungen umsetzen zu können. Wir haben nicht den Anspruch, bereits bestehende Lösungen zu kopieren und dort nach Verbesserungen zu suchen. Daher gibt es auch keinen festen Zeitplan, beispielsweise im Jahr 2010 das stärkste Schachprogramm zu stellen.

Trotzdem verfolgen wir aktuelle Entwicklungen in der Schachprogrammierung und es existiert eine grosse und ständig wachsende Ideenliste, die von Zeit zu Zeit abgearbeitet wird. In naher Zukunft sind für den Schachmotor *DarkFUSc#* grosse Umbauarbeiten am Code geplant. Die Schacherkennung soll optimiert, Suchbaumerweiterungen sowie Endspieldatenbanken eingeführt werden.

Ein Clone mit dem Namen *NeuroFUSc#*, das ein Neuronales Netz (ähnlich wie bei *NeuroChess* [Thrun]) als Bewertungsfunktion verwendet steht kurz vor der ersten Releaseversion. Dabei spielt der Stellungsklassifikator eine sehr grosse Rolle, denn wie in [Block 2004] diskutiert wird, gibt es wichtige Unterschiede zwischen Backgammon (bei dem die Verwendung von Neuronalen Netzen in Backgammonprogrammen dazu führte, dass diese inzwischen sehr viel stärker spielen als menschliche Spieler) und Schach.

Auf dem FUSc#-Server ist die Einführung einer Ligaverwaltung geplant. Mannschaften können sich anmelden und gegeneinander spielen. Es werden neben den bereits existierenden Titeln *FUSc#-Server-Bullet-Meister* und *FUSc#-Server-Blitz-Meister* weitere eingeführt (z.B. *FUSc#-Server-Mannschafts-Meister*). Zur Verwaltung von Mannschaftsturnieren ist ein neuer Serverbot geplant. Dieses Ligasystem soll voll automatisch auf Schachprogramme erweitert werden und so aktuelle Informationen zu den unterschiedlichen Spielstärken geben.

Gambit wird erweitert und liefert Informationen über laufende Partien, so können die Partien auf der Webseite live verfolgt werden.

[7] Deep Shredder 9 von Stefan Meyer-Kahlen ist der aktuelle Weltmeister der Schachprogramme.

8 Anhang

FM Ilja Brener(2509) - **DarkFUSc#**(1842) [15.09.2004] 1/2-1/2

10 Minuten-Blitzpartie

1.d4 Sf6 **2.**c4 g6 **3.**Sc3 Lg7 **4.**e4 0-0 **5.**f3 d6 **6.**Le3 b6 **7.**Dd2 c5 **8.**Ld3 Sg4! **9.**d5 Sxe3 **10.**Dxe3 Sd7 **11.**Sge2 Se5 **12.**0-0 Ld7 **13.**h3 Rc8 **14.**f4 Sxc3 **15.**Dxd3 f5!? **16.**Qd2?! fxe4 **17.**Sxe4 b5!? **18.**cxb5 Lxb5 **19.**Tfe1 Tb8 **20.**S2c3 Ld7 **21.**Sg5 Tb4 **22.**Te4? Lf5? **23.**Txb4 cxb4 **24.**Se2 a5 **25.**Kh1 Dc8?? **26.**Tc1 Dd8? **27.**Sd4 Lf6?? **28.**Sge6 Da8 **29.**Sxf5 gxf5 **30.**Sxf8 Dxf8 **31.**b3 Dh6 **32.**Tf1 Df8 **33.**Dd3 Dc8 **34.**Dc4 Dxc4 **35.**bxc4 a4 **36.**Tb1 Lc3 **37.**g4 fxg4 **38.**hxg4 b3!! **39.**axb3 a3! **40.**b4 a2 **41.**Tc1 a1=D **42.**Txa1 Lxa1 **43.**c5 dxc5 **44.**bxc5 Kf7 **45.**Kg2 e5 **46.**f5 e4 **47.**g5 Ld4 **48.**c6 Le5 **49.**Kf2 Lf4 **50.**d6 Ke8 **51.**g6 hxg6 **52.**fxg6 Lxd6 **53.**Ke3 Le5 **54.**Kxe4 Lg7 **55.**Kf5 Ke7 **56.**c7 Kd7 **57.**c8=D+ Kxc8 **58.**Ke6 Ld4 **59.**Kf7 Kb8 **60.**Ke6 Kc8 **61.**Ke7 Lg7 **62.**Ke8 Le5 **63.**Ke7 Kb8 **64.**Kd7 Lg7 **65.**Ke8 Ld4 **66.**Kd7 Le5 **67.**Ke6 Ld4 **68.**Kd5 Lb2 **69.**Kc4 Le5 **70.**Kd5 Lb2 **71.**Ke6 Ld4 **72.**Kf7 *Remis*

.

Literatur

[Arena-Webseite] Arena-Webseite: *http://www.playwitharena.com/*

[BaxTriWea 1998] Baxter, Jonathan; Triddgell, Andrew; Weaver, Lex: *KnightCap:A chess program that learns by combining TD(lambda) with minimax search*, http://citeseer.ist.psu.edu/baxter98knightcap.html

[Block 2003] Block, Marco: *Reinforcement Learning in der Schachprogrammierung.* Studienarbeit, Freie Universität Berlin 2004

[Block 2004] Block, Marco: *Verwendung von Temporale-Differenz-Methoden im Schachmotor FUSc#.* Diplomarbeit, Freie Universität Berlin 2005

[Buchner 2005] Buchner, Johannes: *Theorie and practical strategies for efficient alpha-beta-searches in computer chess*. Bachelorarbeit, Freie Universität Berlin 2005

[Buchner Sem] Buchner, Johannes: *Rotated bitboards in FUSc#*, Seminararbeit, Freie Universität Berlin 2005

[Chessbase-Webseite] Chessbase-Webseite: *http://www.schach.de*

[CVS-Tutorial] CVS-Tutorial: *http://page.mi.fu-berlin.de/~fusch/fusch/publikation/fuschcvs/index.html*

[Dill Sem] Dill, Sebastian: *Transpositionstabellen*, Seminararbeit, Freie Universität Berlin 2005

[FUSch-Webseite] FUSc#-Webseite: *http://www.fuschmotor.de.vu*

[Fürnkranz 2001] Fürnkranz, Johannes; Kubat, Miroslav: *Machines That Learn To Play Games*, Nova Science Publisher 2001

[Gherrity 1993] Gherrity, Michael: *A Game-Learning Machine*, Dissertation, University of California, San Diego, 1993.

[GNUChess] GNUChess-Webseite: *http://www.tim-mann.org/gnuchess.html*

[Google-Webseite] Google-Webseite: *www.google.de*

[Hyatt-Webseite] Hyatt-Webseite: *http://www.playwitharena.com/*

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

[King 1997] King, Daniel: *Kasparow gegen Deep Blue*, Joachim Beyer Verlag 1997

[McKenzie-Webseite] McKenzie-Webseite: *http://homepages.caverock.net.nz/~peter/perft.htm*

[Mono-Webseite] Mono-Webseite: *http://www.mono-project.com/Main_Page*

[Moreland] Moreland, Bruce: *http://www.seanet.com/~brucemo/topics/0x88.htm*, Homepage von Bruce Moreland

[NProf-Webseite] NProf-Webseite: *http://nprof.sourceforge.net/Site/SiteHomeNews.html*

[Plaat 1996] Plaat, Aske: *RESEARCH, RE:SEARCH & RE-SEARCH*, Dissertation, Tinbergen Institute 1996

[Samuel 1959] Samuel, A.L.: *Some Studies in Machine Learning Using the Game of Checkers*. IBM Journal of Research and Development, 3:210-229, 1959

[SharpDevelop-Webseite] SharpDevelop-Webseite: *http://www.icsharpcode.net/OpenSource/SD/Default.aspx*

[Sutton 1988] Sutton, Richard S.: *Learning to Predict by the Methods of Temporal Differences*. Machine Learning, 3:9-44, 1988

[Tesauro] Tesauro, Gerald: *Comparison Training of Chess Evaluation Functions*, Kapitel 6 in [Fürnkranz 2001]

[Thrun] Thrun, Sebastian: *Learning To Play the Game of Chess,* http://www.cs.cmu.edu/~thrun/papers/thrun.nips7.neuro-chess.ps.gz

[Tortoise-Webseite] Tortoise-Webseite: *http://www.tortoisecvs.org/*

[UCI-Webseite] UCI-Webseite: *http://www.uciengines.de*

[WinBoard-Webseite] WinBoard-Webseite: *http://www.tim-mann.org/xboard.html*