Viscardi-Gymnasium
Fürstenfeldbruck

Oberstufenjahrgang 2013/15

Seminararbeit im Leitfach

Informatik

Optimierung und Laufzeitanalyse einer künstlichen Intelligenz für das Spiel Vier gewinnt

Verfasser: Tobias Hilbig

Kursleiter: StRin Veronika Weichselgartner

Abgabetermin: 04.11.2014

Note: _____

Punktzahl: _____

(einfache Wertung)

_____

(Unterschrift des Kursleiters)

Inhaltsverzeichnis

1 Vorhandenes Vier gewinnt Spiel

Am Ende der zehnten Klasse wurde im Rahmen eines Gruppenprojekts, an dem ich beteiligt war, ein Vier gewinnt Spiel in Java programmiert, welches zwei menschliche Spieler gegeneinander antreten lässt. Es verfügt über eine graphische Oberfläche, die das Spielfeld darstellt. Als Erweiterung wurde die Möglichkeit geschaffen, gegen den Computer, also eine künstliche Intelligenz (KI) zu spielen. Diese ist normalerweise nicht in der Lage, einen menschlichen Gegner zu besiegen. Die KI ist in vier verschiedenen Versionen verfügbar, welche mit unterschiedlichen Schwierigkeitsstufen spielen. Die einfachste Stufe wirft den Stein zufällig in eine freie Spalte ein, die schwerste Stufe lässt sich nur noch durch Zwickmühlen besiegen. Die Verbesserung der KI bot sich als Thema für meine Seminararbeit an, da die Programmierung bisher interessant war und das bestehende Spiel mein Interesse für weitere Entwicklungen geweckt hat. Ziel ist es den Min-Max-Algorithmus so weit zu optimieren, dass ein menschlicher Spieler nur schwer gewinnen kann. Zudem soll der Computer in angemessener Zeit reagieren. Zu Beginn wird die Oberfläche des Spieles erklärt, danach folgt eine Beschreibung des verwendeten Algorithmus, welcher rekursiv arbeitet. Mit den darauf folgenden Optimierungen wird die Laufzeit untersucht und ausgewertet. Zuletzt werden weitere Verbesserungsmöglichkeiten aufgezeigt. Umgesetzt wurde das Programm in Java mit der IDE „Eclipse Kepler Service Release 2“.

2 Realisierung der künstlichen Intelligenz

2.1 Oberfläche

Die graphische Oberfläche des Spieles besteht aus einem Spielfeld mit 6x7 einzelnen Feldern, welche unterschiedlichen Farben darstellen können. Aus ästhetischen Gründen wurde hierbei auf die klassische schwarzweiß Farbgebung verzichtet und stattdessen Rot und Gelb als Spielerfarben gewählt. Über diesen Feldern sind sieben Knöpfe, durch welche der Spieler angeben kann, in welche Spalte sein Stein im nächsten Zug ge-

worfen werden soll. Unter den Feldern ist ein Informationsbereich mit einer Stoppuhr, welche die Spielzeit anzeigt und ein Button mit dem sich jederzeit ein neues Spiel starten lässt. Zusätzlich haben beide Spieler je einen Punktezähler und einen Kreis welcher anzeigt, welche Partei am Zug ist. Unterhalb der Punktezähler ist ein Panel, welches Hinweise und Informationen zum Spiel ausgibt.

Abbildung 1 zeigt die Oberfläche beispielhaft während eines Spieles:

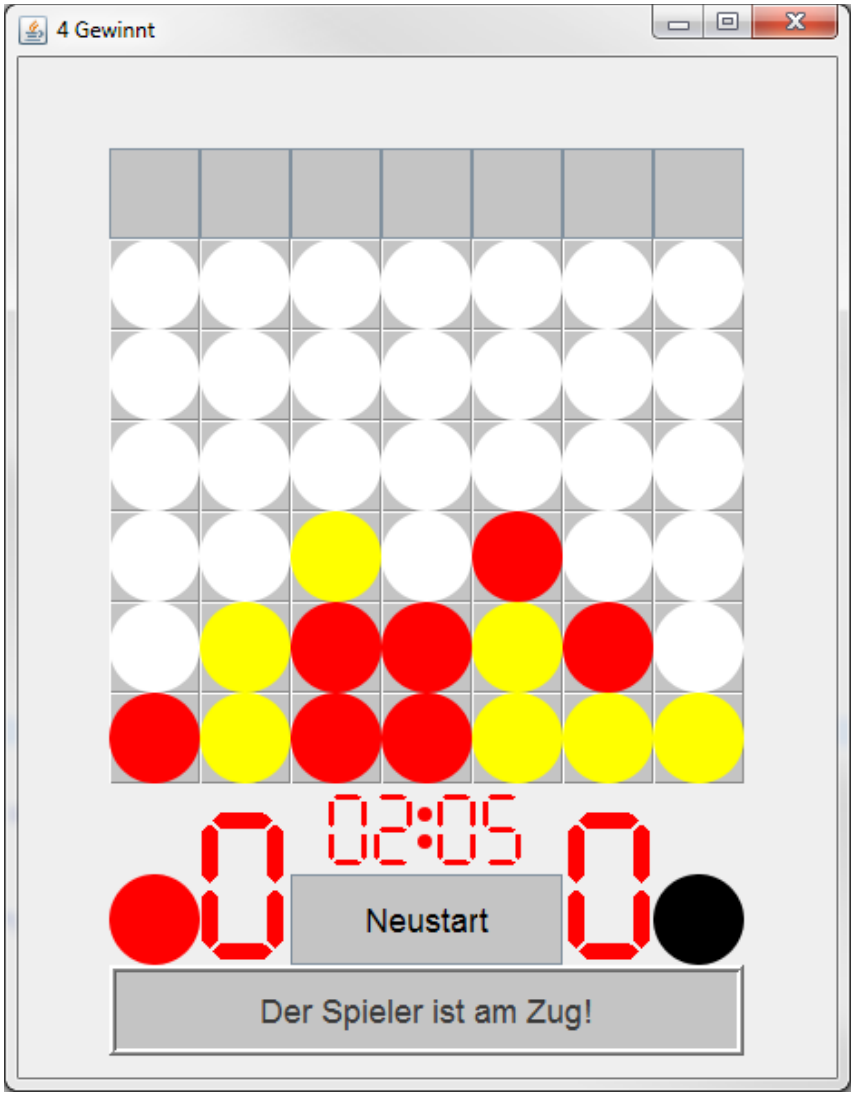

Abbildung 1: Oberfläche des Spiels

Programmiertechnisch ist die grafische Schnittstelle durch Java / AWT realisiert. Die Eingabefelder zu Beginn des Spieles sind in Java / Swing umgesetzt. Alle benötigten graphischen Komponenten werden aus einem Framework namens „Toolbox.java“ eingebunden, welche bereits

vom Informatiklehrer der zehnten Klasse, Herrn Norbert Stamminger, zur Verfügung gestellt wurde. Zur Zeitmessung wurde das Framework „StopWatch.java“ von Corey Goldberg in leicht abgewandelter Form benutzt.

2.2 Vorüberlegung

2.2.1 Ablauf

Am Anfang des Spiels beginnt der menschliche Spieler, in den darauffolgenden Runden darf der Verlierer der vorherigen Runde beginnen. Eine Runde gilt als gewonnen, wenn ein Spieler vier von Steine seiner Farbe in eine Reihe bringt, wobei diese Reihe vertikal, horizontal oder diagonal verlaufen kann. Durch das Drücken eines der sieben Knöpfe oberhalb des Spielfeldes wird der nächste Stein des Spielers in die gewählte Spalte geworfen. Danach ist die künstliche Intelligenz am Zug. Sie bekommt zur Berechnung ihres Zuges die Spielmatrix, ein zweidimensionales Integer-Array, übergeben. Aus dieser Matrix wird die Nummer der Spalte, in die sie einwerfen wird, berechnet. Das Ergebnis wird an das Hauptprogramm zurückgegeben und der Einwurf auf der Oberfläche dargestellt. Dieser Ablauf wird bis zum Ende des Spieles fortgeführt. Für das Spielende gibt es drei mögliche Optionen: Gewinn, Verlust oder Unentschieden.

2.2.2 Ziel

Das Ziel der KI ist es, vor dem Spieler regelkonform zu gewinnen. Zudem muss die Antwortzeit der KI angemessen kurz sein. Optimale Reaktionszeiten sollten daher weniger als zwei Sekunden betragen; akzeptabel sind unter fünf Sekunden. Längere Antwortzeiten sind nicht gewünscht, da der menschliche Gegner sonst zu lange warten müsste.

2.2.3 Grenzen

Durch den begrenzten Arbeitsspeicher des Computers ist ein vollständiges Simulieren aller möglichen Spielsituationen nicht möglich, da der Zustandsraum bei Vier gewinnt zu groß ist. „For the standard, 7 x 6 board, the program found an upper bound of 7.1*10^13“ (Allis: 1988, S. 10.) Nimmt man pro möglichem Spielzustand eine Speichergröße von 3 bit /

Zelle an, so ergeben sich bei einem Standard Spielfeld mit 6*7 Feldern $3*6*7*7{,}1*10^{13}=8{,}946*10^{15}$ bit Speicherbedarf. Das sind etwa 1,1 Petabytes für alle simulierten Zustände. Eine Speicherung der simulierten Züge und Ergebnisse auf Festplatte oder andere permanente Datenträger ist wegen des Zeitaufwands für Lesen und Schreiben prinzipiell nicht vorgesehen.

Außerdem wird die Anzahl der Züge, die die KI vorausberechnen kann, durch weitere Faktoren limitiert. Die Geschwindigkeit der verwendeten Programmiersprache und der konkreten Implementierung, in diesem Fall Oracle Java 1.7.0_67 auf einer 64-bit Maschine, sind entscheidende Faktoren. Zudem spielt die Geschwindigkeit des Prozessors eine Rolle, da das Programm nur eine begrenzte Zeit zur Berechnung des nächsten Zuges hat.

2.3 Implementierung und Optimierung

2.3.1 Beschreibung

Mit dem Min-Max-Algorithmus ist es möglich, bei Spielen, die folgende Eigenschaften besitzen

- zwei Spieler
- endlich, jedes Spiel endet nach maximal 42 Zügen
- perfekte Information, das Spielfeld ist für beide Parteien zu jedem Zeitpunkt sichtbar, es gibt keine verdeckten Informationen
- Nullsumme, daher ist die numerische Summe aller Verluste und Gewinne aller Spieler gleich null.

eine optimale Spielstrategie zu finden[1]. Diese Eigenschaften treffen alle auf das Spiel Vier gewinnt zu. Daher wird dieser Algorithmus für die KI in dieser Anwendung eingesetzt. Der Min-Max-Algorithmus simuliert alle möglichen Züge und durchsucht danach alle Blätter des entstandenen Baumes. Die Blätter werden aus Sicht von Spieler 1 mit -1 für Verlust, 0 für Unentschieden und +1 für Sieg bewertet. Danach werden aus Sicht des gerade Ziehenden der beste Zug gesucht, es wird also maximiert.

1 Mayefsky / Anene / Sirota: 2003, MINIMAX.

Eine Ebene höher wird der schlechteste Zug gesucht, da dieser aus Sicht des Gegners der beste ist, es wird also minimiert. Dieser Vorgang wird bis zur Wurzel des Baumes fortgesetzt. Nachdem alle Knoten durchsucht worden sind, ergibt sich der beste Zug aus der besten Bewertung.

Eine mögliche Nutzung dieses Algorithmus stellt die NegaMax-Suche dar. Für sie wird nur eine gemeinsame Bewertungsfunktion benötigt, die für beide Spieler funktioniert, und bei der jeder Knoten gleich behandelt werden kann. Dafür wird die Bewertung der Stellung immer aus Sicht des ziehenden Spielers maximiert, und der Wert aus einer tiefer liegenden Ebene negiert. Daher auch die Bezeichnung. Somit vereinfacht sich auch der Code, da zwischen eigenen und gegnerischen Knoten nicht mehr unterschieden wird. In Abbildung 4 auf Seite 21 ist die Implementierung der NegaMax-Suche in Pseudo-Code aufgeführt.

Eine solche Suche ist jedoch nicht praktikabel, da sich der Baum bei weitem zu groß ist, siehe Abschnitt 2.2.3 Grenzen. Deswegen ist die Suche nicht für den ganzen Spielbaum durchführbar und wird nur bis zu einer gewissen Tiefe durchgeführt. Durch die begrenzte Tiefe entsteht ein Horizont, hinter den die KI nicht „sehen“ kann. Dadurch kann es vorkommen, dass siegreiche Züge nicht gefunden werden. In der Praxis lässt man die KI nur bis zu einer gewissen Tiefe suchen, um eine angemessen Spielstärke zu erhalten. Dazu wird die Bewertungsfunktion angepasst, sie liefert +∞ falls der ziehende Spieler gewinnt, und -∞ falls der Gegner gewinnt. Ein Unentschieden wird mit 0 bewertet. Alle anderen Stellungen werden heuristisch zwischen +∞ und -∞ eingeordnet. Je tiefer gesucht wird, desto besser kann die KI spielen, und desto schwerer wird es für den menschlichen Gegner zu gewinnen. Wenn das Spiel sich dem Ende nähert, und somit die Anzahl der verbleibenden Züge kleiner als die Suchtiefe des Computers ist, kann die KI perfekt spielen, da alle möglichen Stellungen berechnen werden können.

Der Min-Max-Algorithmus ist zudem vom Spiel unabhängig. Er wird genauso bei Mühle oder Schach eingesetzt, nur die Bewertungs- und Zugfunktionen unterscheiden sich.

2.3.2 Bewertungsfunktion

Die Bewertungsfunktion wird benutzt, um den Wert eines Spielstandes numerisch und heuristisch einzuordnen. Diese Information wird vom Algorithmus benötigt, um Züge zu vergleichen und den besten Zug zu selektieren. Hier liegt der interessanteste Teil des Programms, da diese Funktion die Spielstärke der KI maßgeblich bestimmt. Da die Funktion *bewerten()* an jedem Blatt des Baumes aufgerufen wird, muss sie außerdem sehr schnell arbeiten. Bei der verwendeten Linearkombination, liegt die Geschwindigkeit bei etwa 400.000 Bewertungen pro Sekunde (Siehe Tabelle S. 9) Diese Funktion zählt die Anzahl der Steine und Reihen für jeden Spieler, gewichtet sie und subtrahiert den gegnerischen Wert vom eigenen. Das Ergebnis entspricht der numerischen Bewertung des Blattes. Die Gewichte wurden auf Basis von eignen Tests ermittelt und entsprechen für einen einzelnen Stein dem Wert 1, zwei Steinen nebeneinander dem Wert 15 und drei Steinen in einer Reihe dem Wert 400. Der Wert des Gegenübers wird zum Schluss mit 1,1 multipliziert, da ein materieller Gleichstand auf dem Feld negativ gewertet, und somit möglichst verhindert wird. In Abbildung 2 ist ein Auszug aus der Bewertungsfunktion zu sehen:

```
private static int bewerten(int[][] feld) {

    int single1 = count1Rows(feld, 1);
    int single2 = count1Rows(feld, 2);

    int double1 = count2Rows(feld, 1);
    int double2 = count2Rows(feld, 2);

    int triple1 = count3Rows(feld, 1);
    int triple2 = count3Rows(feld, 2);

    int valueHuman = single1 * gewicht1 + double1 * gewicht2 + triple1
            * gewicht3;

    int valueKI = single2 * gewicht1 + double2 * gewicht2 + triple2
            * gewicht3;

    valueKI = (int) (valueKI * factor);

    return valueHuman - valueKI;
}
```

Abbildung 2: Bewertungsfunktion

Folgende Laufzeiten und Bewertungen ergeben sich abhängig von n:

n	7^n	Bewertungen	%	Zeit (ms)
1	7	7	**100**	0
2	49	49	**100**	0
3	343	343	**100**	1
4	2.401	2.401	**100**	5
5	16.807	14.143	**84,1**	60
6	117.649	97.560	**82,9**	250
7	823.543	575.346	**69,8**	1.730
8	5.764.801	3.908.280	**67,8**	6.698
9	40.353.607	23.139.942	**57,3**	35.747

Es ist klar erkennbar, dass die Rechenzeit von den Bewertungen abhängig ist und mit steigendem n exponentiell wächst. Zudem werden in allen Fällen mindestens 50% aller theoretisch möglichen Zustände simuliert, was höchst ineffizient ist. Da eine schnelle Reaktion des Computers gewünscht ist, sind Tiefen in dieser Version mit n > 7 nicht praktikabel, da die Antwortzeit zu lange ist.

Es ergeben sich nach einer Serie von realen Tests, bei denen der Spieler zuerst werfen durfte, mit n = 7 folgende Ergebnisse: 50 Spiele, davon 3 unentschieden, 15 Siege für den Menschen und 32 Siege für die KI. Dies gibt eine Quote von 64% Siegen für die KI.

Um die Spielstärke zu erhöhen, gibt es einige Verbesserungsmöglichkeiten. Offensichtlich ist eine Erhöhung von n nötig. Dazu muss jedoch die benötigte Zeit sinken. Da sich der Algorithmus selbst nicht optimieren lässt, bleiben vier Möglichkeiten zur Steigerung der Spielstärke übrig:

- Verbessern der Bewertungsfunktion für genauere Ergebnisse
- Verkürzung der benötigten Zeit für die Bewertung
- Verringern der benötigten Vergleiche
- Bessere Ressourcennutzung

2.3.3 Alpha-Beta-Suche

Eine Optimierung zur Verringerung der benötigten Vergleiche ist die Al-

pha-Beta-Suche. Sie stellt eine Erweiterung des Min-Max-Algorithmus dar. Um die Anzahl der untersuchten Blätter zu reduzieren werden zwei neue Variablen genutzt, *alpha* und *beta*. Diese stellen das Worst-Case-Szenario beider Spieler dar. Zu Beginn der Suche sind *beta* = +∞ und *alpha* = -∞. Im weiteren Verlauf der Suche werden diese Werte angepasst und nur noch in dem aus den Werten entstandenen Fenster gesucht („Cutoff")[2]. Werden Knoten mit Rückgabewerten außerhalb dieses Fensters gefunden, so werden weitere Züge dieses Astes nicht untersucht, da sie aufgrund ihrer zu schlechten, oder aus Sicht des Gegners, zu guten Bewertung nicht in für den Spieler in Frage kommen. Bei der Implementierung in Form einer NegaMax-Suche werden die beiden Werte bei der Rekursion vertauscht und negiert, da aus Sicht des Gegners gesucht wird. Der Alpha-Beta-Algorithmus ist im Anhang in Abbildung 4 dargestellt, wobei die Erweiterungen zum Min-Max-Algorithmus gelb hinterlegt sind.

Folgende Laufzeiten und Bewertungen ergeben sich bei Optimierung durch Alpha-Beta abhängig von *n*:

n	Bewertungen Min-Max	Bewertungen Alpha-Beta	%	Zeit (ms) Min-Max	Zeit (ms) Alpha-Beta	%
5	14.143	5.000	**35,3**	60	33	**55,0**
6	97.560	16.000	**16,4**	250	60	**24,0**
7	575.346	43.000	**7,4**	1.700	120	**7,0**
8	3.908.280	270.000	**6,9**	11.900	600	**5,0**
9	23.139.942	750.000	**3,2**	80.000	1.900	**2,3**
10	*	5.100.000	**-**	*	11.900	**-**
11	*	10.200.000	**-**	*	25.750	**-**
12	*	105.000.000	**-**	*	230.000	**-**

** Berechnung aufgrund des Zeitaufwands nicht durchgeführt. Anzumerken ist hierbei, dass an die Bäume der Ebene n = 0 keine bereits berechneten alpha und beta Werte weitergegeben werden. Dies verlängert die Suchzeit, ist aber gewünscht, um den Fortschritt grob ausgeben zu können. Zudem ist eine spätere Nutzung von Multithreating möglich. Werte teilweise gerundet.*

Die Anzahl der Bewertungen und somit der Zeitaufwand haben sich in dieser Version bei n = 9 etwa um Faktor 50 verringert. Es ist erkennbar,

2 Mayefsky /Anene / Sirota: 2003, ALPHA-BETA PRUNING.

dass die Effizienz mit steigendem n zudem noch größer wird. Durch das Abschneiden von nicht relevanten Teilbäumen ändert sich nur die Laufzeit und nicht die Spielstärke. Mit dieser Optimierung sind nun Spiele mit n = 9 praktikabel, da die Antwortzeit stark gesunken ist. Die Erhöhung von n verbessert die Spielstärke der künstlichen Intelligenz.

Es ergeben sich nach einer Serie von realen Tests, bei denen der Spieler zuerst werfen durfte, mit n = 9 folgende Ergebnisse: 50 Spiele, davon 3 unentschieden, 3 Siege für den Menschen und 44 Siege für die KI. Dies gibt eine Quote von 88% Siegen für die KI.

2.3.4 Zugsortierung

Bisher wurden die Züge der Reihe nach von 1 bis 7 ausgewertet. Eine schnelle Alpha-Beta-Suche benötigt jedoch sortierte Züge, damit das Suchfenster möglichst früh verkleinert werden kann. Werden gute Züge zu Beginn der Suche gefunden, so müssen im weiteren Verlauf weniger Äste untersucht werden. Dazu müssen die Züge nach ihrem vermuteten Wert vorsortiert werden. Die theoretische, maximale Anzahl der Vierer-ketten, die durch das entsprechende Feld gelegt werden können, stellt einen einfachen aber guten Sortierschlüssel dar. Hierbei kann man wie bei „Row Sort“ nur die Spalte betrachten und somit Züge in die mittleren Spalten bevorzugen. Wertet man jedes Feld einzeln, ist die Qualität der Sortierung höher. Dies wir bei „Matrix Sort“, mithilfe eines statischen Array[3] erreicht. Dieses bewertet Züge nach dem Wert des Feldes, in das der geworfene Stein fallen wird. Die Gewichtungen für die Sortierung beider Verfahren sind in den folgenden Tabellen dargestellt:

Gewichtung der Spalten bei Row Sort:

Spalte	1	2	3	4	5	6	7
Gewichtung	1	2	3	4	3	2	1

Gewichtung der Felder bei Matrix Sort:

3	4	5	7	5	4	3
4	5	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	5	8	10	8	6	4
3	4	5	7	5	4	3

3 Baier: 2006, S. 46.

Dabei ergeben sich folgende Laufzeiten und Bewertungen abhängig von *n:*

n	Bewertungen x 1000					Zeit (ms)				
	Keine Sort.	**Row Sort**	**%**	**Matrix Sort**	**%**	**Keine Sort.**	**Row Sort**	**%**	**Matrix Sort**	**%**
5	5	3,3	**66**	3,4	**68**	33	31	**93**	21	**63**
6	16	9,2	**57**	10,8	**67**	60	59	**98**	61	**101**
7	43	21	**48**	24,9	**57**	120	91	**89**	131	**109**
8	270	108	**40**	138	**51**	600	270	**45**	391	**65**
9	750	243	**32**	290	**38**	1.900	660	**34**	931	**49**
10	5.100	1.280	**25**	2.400	**47**	11.900	3.900	**32**	6.600	**55**
11	10.200	3.172	**31**	3.700	**36**	25.750	8.300	**32**	12.700	**49**
12	105.000	31.000	**29**	41.000	**39**	230.000	66.000	**28**	106.300	**46**

Auch hier werden die errechneten Alpha und Beta Werte nicht in Ebene n = 0 genutzt. Siehe S. 10.
In Ebene 0 wird zudem keine Sortierung angewendet.
Werte teilweise gerundet.

Mithilfe der Zugsortierung lassen sich nochmals erhebliche Geschwindigkeitssteigerungen erzielen. Bei Row-Sort beträgt der Faktor der Beschleunigung Faktor 3, bei Matrix Sort bis Faktor 2. Die Differenz lässt sich durch den erhöhten Zeitaufwand für die Entnahme des Wertes aus der Matrix schließen. Da die Zugsortierung keinen Einfluss auf die Spielstärke, sondern nur auf die benötigte Zeit hat, wird für weitere Optimierungen die schnellere Variante benutzt. Spiele mit n > 9 sind auch mit Zugsortierung nicht durchgehend möglich, da die Antwortzeit zum Teil inakzeptabel bleibt.

2.3.5 Multithreating

Um die Performance weiter zu erhöhen, kann bei Computern mit Mehrkernprozessoren auf mehreren Rechenkernen gleichzeitig gerechnet werde. Der Min-Max-Algorithmus ist nur mit Performanceminderung parallelisierbar, da die einzelnen Bäume voneinander abhängig sind. Die Alpha Beta Werte können während der Berechnung nicht zwischen den einzelnen Threads synchronisiert werden. Dadurch müssen mehr Bewertungen vorgenommen werden. Dieser Zeitverlust ist jedoch auch in allen vorherigen Tests vorhanden gewesen, da alpha und beta in Ebene n=0

nicht genutzt wurden. Somit hat die Nutzung von Multithreating keinen Einfluss auf die Anzahl der nötigen Bewertungen.

Folgende Laufzeiten ergeben sich abhängig von t:

Threads (t)	**Zeit (s) n = 11**	**% von t = 1**
1	10,2	**100,0**
2	5,3	**51,9**
3	3,6	**35,2**
4	3,5	**34,3**
5	3,2	**31,3**
6	3,1	**30,3**
7	3,0	**29,4**

Auch hier werden die errechneten Alpha und Beta Werte nicht in Ebene n = 0 genutzt. Siehe S. 10.
In Ebene 0 wird die Sortierung nicht angewendet.
„Zeit(s)“ auf 0,1s gerundet.
Prozentwerte auf 1 Nachkommastelle gerundet.

Durch die Nutzung von Multithreating lässt sich die Performance nochmals um Faktor 3 steigern. Die theoretische, maximale Steigerung der Geschwindigkeit liegt durch die Möglichkeit der Parallelisierung in meinem Fall bei Faktor 4. Der Unterschied zu den realen Ergebnissen wird durch Aufwand für Verwaltung, Synchronisierung und andere Anwendungen generiert. Zudem kann es vorkommen, dass einzelnen Threads mehr Zeit benötigen, was sich durch die nötige Wartezeit negativ auf die Gesamtzeit auswirkt. Alle bisher durchgeführten Optimierungen haben keinen Einfluss auf die Spielstärke, da nur die Antwortzeit des Programms verkürzt wird. Dies lässt sich einfach durch vier parallel gestartete Spiele mit je einer eigenen Optimierung zeigen: Alle kommen zum exakt selben Ergebnis. Die Spielstärke wird nur dadurch erhöht, dass größere Suchtiefen möglich sind. In Abbildung 3 ist das Flussdiagramm der Threadverteilung zu sehen. Diese arbeitet unabhängig von der Zahl der Prozessorkerne und verteilt die anfallende Arbeit, in diesem Fall die Berechnung der Züge, auf die vorhandenen Prozessorkerne. Während der Berechnung wird der Fortschritt in Prozent an das Hauptprogramm weitergegeben und dort dargestellt.

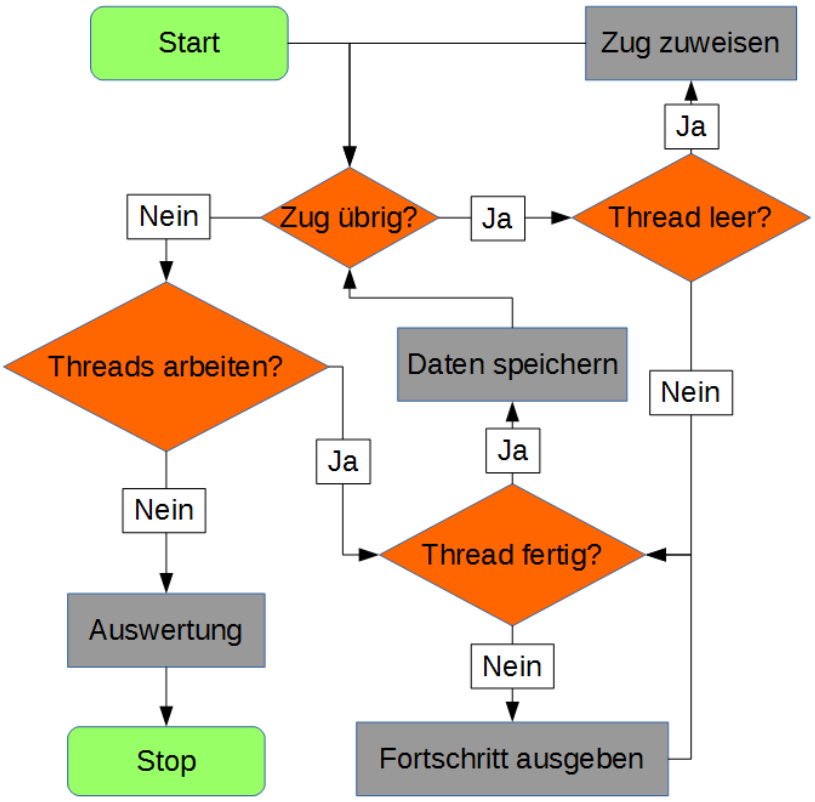

Abbildung 3: Threadverteilung

Mit Multithreating ist auf einer 4-Kern CPU nun sogar Tiefe n = 11 bei akzeptabler Reaktionszeit spielbar. Es ergeben sich nach einer Serie von realen Tests, bei denen der Spieler zuerst werfen durfte, mit n = 11 folgende Ergebnisse: 50 Spiele, davon 1 unentschieden, 2 Siege für den Menschen und 47 Siege für die KI. Dies gibt eine Quote von 94% Siegen für die KI. Wie in Abschnitt 2.5.4 Rechenzeit beschrieben, gibt es noch einige weitere Möglichkeiten, eine künstliche Intelligenz für das Spiel Vier gewinnt zu verbessern. Da die vorliegende Implementierung eine hinreichende Spielstärke bei akzeptabler Antwortzeit garantiert, wurde auf weitere Optimierungen verzichtet.

2.4 Laufzeitanalyse

2.4.1 Allgemeines

Die Laufzeit des Codes wurde mit jeder Verbesserung erneut getestet

und ausgewertet. Die Ergebnisse wurden in dieser Seminararbeit dargestellt. Da die benötigte Zeit mit zunehmender Tiefe n exponentiell steigt, wurde die Zeit in Abhängigkeit der Tiefe angegeben. Die Berechnungen wurden, soweit nicht anders angegeben, auf einem Intel Core i7 4790K Prozessor mit 8 Threads und 4 Kernen zu je 4,0 GHz durchgeführt.

2.4.2 Ergebnis

Der Vergleich zwischen der ersten, nicht optimieren Min-Max Version und der optimierten, sortierten Alpha-Beta-Suche zeigt sehr deutlich, wie stark die Optimierungen sind.

Folgende Bewertungen und Zeiten ergeben sich bei Tiefe n = 11:

Version	Bewertungen	Zeit (s)	% von Min-Max
Min-Max	884.000.000	1.536,0	**100,000**
Alpha-Beta	10.000.000	25,0	**1,627**
Alpha-Beta Row Sort	3.000.000	10,8	**0,703**
Multithreted	3.000.000	3,1	**0,201**

„Bewertungen“ auf 1 Million gerundet.
„Zeit(s)“ auf 0,1s gerundet.
Prozentwerte auf 3 Nachkommastellen gerundet.

2.5 Verbesserungsmöglichkeiten

2.5.1 Spielstärke

Die verwendete Bewertungsfunktion stellt eine einfache Linearkombination dar, welche zwar schnell arbeitet, jedoch bestimmte Eigenschaften von Spielsituationen außer Acht lässt. Es gibt Situationen, in denen eine Reihe aus drei eigenen Steinen nicht siegen kann, diese darf somit nicht positiv in die Bewertung eingehen. Zum Beispiel können drei eigene Steine in einer Reihe liegen, die an beiden Enden von gegnerischen Steinen oder dem Spielfeldrand blockiert werden, und somit keinen Gewinn erzielen. Dies kann verbessert werden, indem die Bewertung nicht statisch auf die gegebene Spielsituation angewendet wird, sondern dynamisch Zukunftsaussichten bewertet. Hierbei ist es jedoch sehr wichtig, zwischen besserer Spielstärke und Laufzeit abzuwägen. Nimmt man eine hohe Laufzeit für eine bessere Spielstärke in Kauf, so kann es vorkom-

men dass ein schlechter, dafür schneller spielender Algorithmus in der selben Zeit bessere Ergebnisse liefert. Daher müssen bei solchen Verbesserungen KI vs. KI Tests durchgeführt werden, um herauszufinden, ob die spielerische Verbesserung nicht durch zu lange Rechenzeit zunichte gemacht wird.

Ein anderer Ansatz ist die Verwendung der Technik „Iterative Deepening“, bei der eine feste Antwortzeitspanne vorgegeben wird und die künstliche Intelligenz innerhalb dieser antworten muss. Um diese Bedingung zu erfüllen, wird die Zugberechnung iterativ mit zunehmendem n durchgeführt. Ist die gegeben Zeit abgelaufen, wird das Ergebnis aus der zuletzt vollständig berechneten Tiefe zurückgegeben. Ein weiterer Vorteil dieser Technik ist die erhöhte Spielstärke gegen Ende des Spiels. Der Spielbaum wird auf Grund von gefüllten Spalten immer schmaler und erlaubt es deshalb, bei gleichem Zeitaufwand tiefer zu suchen. Zudem können Ergebnisse aus der Tiefe n-1 für die Sortierung der Züge in Tiefe n genutzt werden, was die Suche zusätzlich beschleunigen würde.

2.5.2 Datenstruktur

Zur Speicherung des Spielfeldes wurde durchgehend ein zweidimensionales, 7x6 Felder großes Integer-Array verwendet, welches den Zugriff auf die Daten vereinfacht, da ein Abruf mittels Kontrollstrukturen und Wiederholungen mit fester Anzahl möglich ist. Eine andere Methode sind Bitboards[4], die eine weitere Steigerung der Performance um bis zu 70% versprechen. Bei dieser Datenstruktur werden die eigenen und gegnerischen Chips in je einem long-Wert gespeichert. Auf diesen Datentyp ist ein schnellerer und effizienterer Zugriff möglich. Die Implementierung ist jedoch weniger einfach, da der Zugriff bitweise erfolgen muss, weswegen dieser Ansatz in dieser Arbeit nicht verwendet wird. Aufgrund der Geschwindigkeitsvorteile sind Bitboards inzwischen in der Schachprogrammierung Standard.

2.5.3 Speicher

Ein Zwischenspeichern der Züge, Knoten und Blätter über einen Zug hin-

4 Baier: 2006, S. 34.

aus findet bei dieser KI aufgrund von begrenztem Hauptspeicher nicht statt, eine solche Speicherung wäre aber grundsätzlich möglich. Der Suchbaum müsste mit den errechneten Stellungswerten gespeichert werden. Hierbei wäre zu beachten, dass nur relevante Knoten gespeichert werden müssen. Somit könnte im nächsten Zug auf bereits errechnete Daten des vorherigen Zuges zurückgegriffen werden, was die Suche beschleunigen könnte.

2.5.4 Rechenzeit

Die Verkürzung der Rechenzeit bis Tiefe n ist durch mehrere Optionen möglich. An allen Knoten des Baumes wird die Funktion *auswerten()* genutzt. Sind die Blätter des Suchbaumes erreicht, wird die Funktion *bewerten()* aufgerufen. Da die rekursiven Vergleiche innerhalb des Baumes aus Sicht der Rechenzeit vernachlässigbar sind, muss sich eine Optimierung auf die beiden angeführten Funktionen beziehen.

Die Auswertungsfunktion lässt sich durch geeignete Optimierung der Schleifenstruktur verbessern. Ein manuelles Testen aller möglicher Vierer-Ketten hätte nur dann Sinn, falls der Compiler die zweifach verschachtelten Schleifen nicht vollständig auflöst. Ob eine solche Auflösung stattfindet, muss durch geeignete Versuche festgestellt werden.

Die Optimierung der Bewertungsfunktion ist nur bedingt möglich, da diese auf einer einfachen Linearkombination basiert und auch in dieser verschachtelte Schleifen genutzt werden. Eine qualitativ hochwertigere Gewichtung wäre ein möglicher Ansatz. Durch Simulation und Auswertung einer entsprechend großen Anzahl von Spielen können optimierte Gewichte gefunden werden.

Die letzte Option ist das Verringern der nötigen Knoten, was mit Optimierungen wie einer noch besseren Sortierung, der Principal-Variation-Suche[5], Aspiration windows[6] oder der Ruhesuche[7] möglich wäre. Für diese Optimierungen gibt es bereits Ergebnisse in Hendrik Baier's Bachelorarbeit.

5 Baier: 2006, S. 20.
6 Baier: 2006, S. 30.
7 Baier: 2006, S. 26.

3 Weitere Ansätze und Möglichkeiten

Der Gegenstand dieser Arbeit ist die Optimierung des Min-Max-Algorithmus für eine künstliche Intelligenz für das Spiel Vier Gewinnt, die ohne Vorwissen, mit akzeptabler Antwortzeit, gegen einen menschlichen Gegner antritt und diesen möglichst besiegt. Dieses Ziel wurde in der letzten, optimierten Version erreicht. Diese künstliche Intelligenz rechnet elf Züge voraus und antwortet in etwa drei Sekunden. Eine Siegesquote von 94% ist ein akzeptables Ergebnis. Der Spieler hat den subjektiven Eindruck, gegen einen sehr guten Gegner zu spielen. Andere Ansätze für eine KI für Vier gewinnt verwenden Datenbanken und permanenten Speicher.

Durch die Nutzung von Datenbanken ist es möglich, in Echtzeit gegen einen perfekt spielenden Computer anzutreten. Dieser kann immer mindestens ein Unentschieden, falls er beginnt, sogar einen Sieg erzwingen[8]. Eine solche KI benötigt jedoch eine vorher berechnete Datenbank mit mehreren Millionen Stellungen.

Eine andere Möglichkeit ist die Verwendung von künstlichen neuronalen Netzen, die selbstständig lernend eine Strategie entwickeln. Dieser Ansatz ist nicht mehr mit dem vorgestellten oder dem von Victor Allis vergleichbar, dar seine Zugwahl nicht auf heuristischen Bewertungen bzw. in Datenbanken gespeicherten Werten basiert, sondern auf selbst gelernter Erfahrung.

8 Allis: 1988, ABSTRACT, S. 1.

Literaturverzeichnis

- [1] Allis, Victor, A Knowledge-based Approach of Connect-Four, 1988, aufgerufen am 11.09.2014, http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf
- [2] Baier, Hendrik, Der Alpha-Beta-Algorithmus und Erweiterungen bei Vier Gewinnt, 2006, abgerufen am 22.08.2014, http://www.ke.tu-darmstadt.de/lehre/arbeiten/bachelor/2006/Baier_Hendrik.pdf
- [3] Eric Mayefsky, Francine Anene, Marina Sirota, ALGORITHMS - ALPHA-BETA PRUNING, http://web.stanford.edu/~msirota/soco/alphabeta.html. 2003, abgerufen am 28.10.2014
- [4] Eric Mayefsky, Francine Anene, Marina Sirota, ALGORITHMS - MINIMAX, http://web.stanford.edu/~msirota/soco/minimax.html. 2003, abgerufen am 18.10.14
- [5] Kopelke, Florian, Selbstständiges Erlernen einer Vier-gewinnt-Spielstrategie durch den NAO-Roboter unter Verwendung künstlicher neuronaler Netze, 2013, aufgerufen am 11.09.2014, http://emt.h-brs.de/emtmedia/Downloads/personen/Rothe/MasterKoplke.pdf

Abbildungsverzeichnis

Alle verwenden Abbildungen sind eigene Werke.

Anhang: Alpha-Beta-Suche mit Unterschied zum NegaMax-Algorithmus

```
static int NegaMaxAB(int[][] feld, int tiefe, boolean spieler,
	int alpha, int beta) {

	int best = NegINF;
	best = alpha;

	int winner = auswertung(feld);

	if (tiefe == 0 || Feld.istVoll() || winner != 0) {
		return bewerten(feld, spieler, winner);
	}

	Move[] moves = generateMoves(feld, spieler);

	for (Move currentMove : moves) {
		werfen(feld, currentMove);

		int value = -miniMaxCutSort(feld, tiefe - 1, !spieler,
-beta, -best);

		fangen(feld, currentMove);

		if (value >= best) {
			best = value;
			if (best >= beta) {
				break;
			}
		}
	}
	return best;
}
```

Abbildung 4: Alpha-Beta-Suche mit Unterschied zum NegaMax-Algorithmus

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel verwendet habe.

Fürstenfeldbruck, 04.11.2014

_____