Implementierung eines Congo-Spielers

Andreas Knab

15. Mai 2007

Studienarbeit von Andreas Knab
Abteilung für Grundlagen der Künstlichen Intelligenz
Institut für Informatik
Albert-Ludwigs-Universität Freiburg
2007

Inhaltsverzeichnis

1 Einleitung

Ein sehr interessantes Teilgebiet der Künstlichen Intelligenz bilden die strategischen Spiele. In dieser Studienarbeit wird das Zweipersonenspiel *Congo* betrachtet, das eine Variante des Klassikers *Schach* ist. Bei dem Spiel handelt es sich – genau wie bei *Schach* – um ein total beobachtbares strategisches Spiel. Das bedeutet, beiden Spielern liegen alle Informationen über die Spielumgebung vor und der Ausgang ist nur von den Zügen der beiden Spieler abhängig, er beinhaltet keine Zufallskomponente. Es ist theoretisch in jeder Situation möglich zu entscheiden, welcher Zug optimal ist – z.B. mittels eines *Minimax*-Algorithmus, wie von Russell und Norvig vorgestellt [9]. Dabei ergibt sich jedoch genau wie bei *Schach* das Problem, dass der Suchraum, auf Grund des durchschnittlichen Verzweigungsgrades von zirka 24, den *Congo* hat, sehr groß wird. Bereits bei einer Suchtiefe von nur acht Halbzügen[1] sind im Schnitt mehr als 10^{11} Zustände zu explorieren, was je nach Implementierung einen kaum zu bewältigenden Zeit- bzw. Speicheraufwand darstellt. Aus diesem Grund kann das Durchsuchen des Suchraums in der Regel nur bis zu einer gewissen Tiefe erfolgen, in der dann die Zustände in irgendeiner Weise bewertet werden müssen.

Im Folgenden werden die Regeln von *Congo* vorgestellt werden, wonach dann einige Ansätze zum Entwurf eines Congo-Programms mit Hilfe von Methoden aus der Künstlichen Intelligenz erläutert werden. Dabei wird als Grundlage der Minimax-Algorithmus gewählt, von dem ausgehend dann die $\alpha\beta$-Suche [9] eingeführt wird. Im Anschluss wird noch gezeigt, wie man mittels *Move-Ordering* und *Transpositionstabellen* die Laufzeit der $\alpha\beta$-Suche weiter verbessern kann. Außerdem werden verschiedene Bewertungsfunktionen für Zustände während des Spiels vorgestellt.

Zum Abschluss werden die Ergebnisse diskutiert, die sich aus der Implementierung von verschiedenen Algorithmen ergeben haben. Dabei werden unter anderem die Laufzeiten verglichen, sowie darauf eingegangen, wie gut die einzelnen Algorithmen und Bewertungsfunktionen im direkten Vergleich gegeneinander abgeschnitten haben.

2 Grundlagen des Congo-Spiels

Das Brettspiel *Congo* wurde 1982 von Demian Freeling, dem damals siebenjährigen Sohn von Christian Freeling, einem niederländischen Spieledesi-

[1] bei Spielen wie *Congo* oder *Schach* versteht man unter einem *Halbzug* die Aktion eines Spielers, während ein Zug aus zwei Halbzügen besteht, also aus je einem Halbzug jedes Spielers

gner, erfunden [4]. Christian Freeling hatte sich bis zu dem Zeitpunkt bereits durch einige andere Schachvarianten auf diesem Gebiet einen Namen gemacht, und er hat wohl auch seinem Sohn bei der Entwicklung von *Congo* tatkräftig zur Seite gestanden.

2.1 Spielidee von *Congo*

Congo hat das gleiche Spielprinzip wie *Schach*: Zwei Spieler ziehen abwechselnd jeweils eine ihrer Figuren, wobei das Ziel darin besteht, den Löwen – dieser entspricht dem König beim *Schach* – des Gegners zu schlagen. Es wird solange gespielt bis einer der Löwen geschlagen ist, das heißt, es gibt kein Unentschieden und ein Spiel könnte theoretisch unendlich lange dauern. Im Gegensatz zu *Schach* ist es auch nicht verboten, sich selbst „matt“ zu setzen; man darf also den eigenen Löwen auf ein Feld bewegen, auf dem er im nächsten Zug geschlagen werden kann.

In *Congo* werden die Spielfiguren fast ausschließlich durch afrikanische Tiere dargestellt, die sich in erster Linie durch andere Bewegungsoptionen von herkömmlichen Schachfiguren unterscheiden. Gespielt wird auf einem sieben mal sieben Felder großen Spielbrett, wobei die Spielfiguren der Spieler zu Beginn eines Spiels genau wie bei *Schach* jeweils in den beiden hinteren Spielbrettreihen aufgestellt sind. Ein weiterer Unterschied zu *Schach* ist, dass die mittlere Reihe auf dem Congo-Brett einen Fluss darstellt, für den noch weitere Bewegungsregeln gelten (vgl. Abschnitt 2.3), ähnlich denen anderer Schachvarianten, wie zum Beispiel im chinesischen Xiangqi [5]. Außerdem wurde noch die Bewegungsfreiheit des Löwen auf ein jeweils drei mal drei Felder großes Gebiet eingeschränkt, welches in Abb.1 dunkelgrau hervorgehoben ist.

2.2 Die Spielfiguren

Bei *Congo* gibt es von jeder Farbe jeweils acht verschiedene Spielfiguren. Der Löwe ist die zentrale Figur, die es zu schlagen gilt. Die weiteren Figuren, die sich zu Spielbeginn auf dem Brett befinden, sind von jeder Farbe die Giraffe, der Affe, jeweils zwei Elefanten, das Krokodil, das Zebra und sieben Bauern, die wie in Abb.1 aufgestellt sind. Ferner kann während des Spiels ein Bauer zu einem Superbauern befördert werden, indem man mit einem Bauern die Grundlinie des Gegners erreicht.

Hier folgt nun eine Auflistung aller Spielfiguren von *Congo* und ihrer Bewegungsmöglichkeiten. Die Buchstaben in Klammern stehen dabei für die englischen Abkürzungen der Figuren, die später für ihre Darstellung auf dem

Abbildung 1: Das Congo-Spielbrett [4]

Spielbrett verwendet werden.

Giraffe (g): Sie kann sich ein oder zwei Felder in eine beliebige Richtung bewegen. Dabei kann sie auch eine Figur überspringen. Schlagen kann sie jedoch nur Figuren, die zwei Felder entfernt stehen.

Affe (m): Er kann sich ein Feld in eine beliebige Richtung bewegen, wenn das jeweilige Feld leer ist. Ferner kann er Figuren schlagen, indem er über sie hinweg auf ein freies Feld hüpft. Hat er dies mit einer Figur getan, kann er das direkt im Anschluss beliebig oft wiederholen – vergleichbar mit einer Dame im Spiel *Dame*.

Elefant (e): Ein Elefant kann sich ein oder zwei Felder in senkrechter oder waagrechter Richtung bewegen und dabei auch eine Figur überspringen. Dabei schlägt er gegnerische Figuren, die auf dem Feld stehen, auf das er zieht.

Löwe (l): Er kann sich wie der Schach-König ein Feld in eine beliebige Richtung bewegen und dabei auch feindliche Figuren schlagen; dabei muss er aber in seinem neun Felder großen Bereich bleiben (grau unterlegt in Abb.1. Es gibt nur eine Ausnahme, bei der er diesen Bereich verlassen darf, und zwar genau dann, wenn er sich mit dem gegnerischen Löwen auf einer Linie (senkrecht oder diagonal) befindet und keine weitere Figur dazwischen steht. In diesem Fall kann er sofort den anderen Löwen schlagen und somit das Spiel beenden.

Krokodil (c): Das Krokodil kann sich ein Feld in eine beliebige Richtung bewegen und dabei auch schlagen. Außerdem kann es sich beliebig viele Felder senkrecht in Richtung des Flusses (und hinein) bewegen – wie ein Turm bei *Schach*. Befindet sich das Krokodil schon im Fluss, dann kann es sich innerhalb des Flusses ebenfalls wie ein Schach-Turm bewegen.

Zebra (z): Das Zebra bewegt sich ganz genau wie ein Springer bei *Schach*, das heißt zwei Felder in eine beliebige Richtung und ein Feld senkrecht dazu oder umgekehrt. Dabei kann es Figuren überspringen und eine gegnerische Figur schlagen, falls diese auf dem Feld steht, auf dem das Zebra seinen Zug beendet.

Bauer (p): Ein Bauer kann sich ein Feld senkrecht oder diagonal nach vorne bewegen und dabei auch schlagen. Befindet sich ein Bauer bereits auf der anderen Seite des Flusses, so kann er sich noch zusätzlich ein oder zwei Felder senkrecht rückwärts bewegen, ohne dabei jedoch Figuren zu schlagen oder zu überspringen.

Erreicht ein Bauer die gegnerische Grundlinie, so verwandelt er sich in einen *Superbauern*.

Superbauer (b): Zusätzlich zu den normalen Bewegungsoptionen eines gewöhnlichen Bauern, kann sich ein Superbauer ein Feld waagrecht zur Seite bewegen und dabei auch schlagen. Außerdem kann er sich – unabhängig von seiner Position auf dem Spielbrett – ein oder zwei Felder senkrecht oder diagonal nach hinten bewegen, ohne dabei jedoch zu schlagen oder zu springen.

2.3 Der Fluss

Der Fluss besteht aus den sieben Feldern in der vierten Zeile des Spielbretts und trennt quasi das Feld in eine „weiße“ und einen „schwarze“ Seite. Er stellt eine Art Hindernis dar – wird nämlich eine Figur in den Fluss bewegt und nicht gleich in der nächsten Runde wieder daraus entfernt, so ertrinkt sie und wird vom Spielbrett genommen. Die einzige Ausnahme von dieser Regel bilden die Krokodile – sie können beliebig lang im Fluss verweilen, ohne dass sie ertrinken.

3 Grundlagen für ein Congo-Programm

Bei *Congo* handelt es sich um ein strategisches Spiel mit totaler Beobachtbarkeit, d.h. es ist allen Spielern zu jeder Zeit möglich, die komplette für das

Spiel relevante Umgebung vollständig zu überblicken – im Unterschied zum Beispiel zu vielen Kartenspielen, bei denen nicht bekannt ist, welche Karten andere Spieler besitzen. Aus diesem Grund ist der erstmal naheliegendste Ansatz zur Implementierung eines Agenten für das Spiel, die Verwendung eines *Minimax*-Algorithmus. Darauf aufbauend kann man dann weitere Algorithmen wie etwa eine $\alpha\beta$-Suche implementieren.

Bevor man aber damit anfangen kann, muss man sich zunächst Gedanken darüber machen, wie das Spielbrett, die Figuren und ganze Spielbrett-Zustände repräsentiert werden sollen. Diese Probleme werden im Folgenden behandelt.

3.1 Repräsentation von Zuständen

Um einen Zustand des Spielbretts darstellen zu können, definieren wir uns zu Beginn ein Array der Größe 49, durch das die Felder des Spielbretts repräsentiert werden. Für die einzelnen Spielfiguren verwenden wir ihre englischen Anfangsbuchstaben als Abkürzung (vgl. Abschnitt 2.2). Zur Unterscheidung zwischen den Farben verwenden wir für schwarze Figuren Kleinbuchstaben und für weiße Großbuchstaben. Als nächstes wird noch gespeichert, welcher Spieler als nächstes am Zug ist, und schon kann man jeden beliebigen Zustand darstellen.

Die erweiterten Eigenschaften des Spielbretts, das heißt der Fluss und die Bewegungsbereiche der Löwen, müssen an dieser Stelle nicht berücksichtigt werden, da sie konstant und damit also nicht relevant für die Beschreibung eines Zustandes sind. Um sie muss man sich erst bei der Berechnung der möglichen Nachfolgezustände eines gegebenen Zustandes kümmern.

Damit ein Zustand für uns etwas besser lesbar wird, wollen wir folgende Darstellung eines Zustandes für Ein- und Ausgabe verwenden:

```
:w
G M E L E C Z
P P P P P P P
. . . . . . .
~ ~ ~ ~ ~ ~ ~

. . . . . . .
p p p p p p p
g m e l e c z
```

Abbildung 2: Darstellung eines Zustands am Beispiel des Startzustands

In der ersten Zeile steht die Farbe des Spielers, der als nächstes am Zug ist – hier `:w` für *white*, alternativ `:b` für *black*. Im Anschluss folgt das Spielbrett. Die Punkte stellen gewöhnliche leere Felder dar, während leere Flussfelder durch `~` symbolisiert werden.

3.2 Berechnung von möglichen Nachfolgezuständen

Auf diese Art und Weise können nun alle möglichen Zustände des Spielbretts dargestellt werden. Um einen Algorithmus zu schreiben, der möglichst gute Züge macht, ist es als nächstes erforderlich, die Zugregeln des Congo-Spieles zu implementieren. Dazu schreibt man eine Funktion `move`, die zu einer Figur, die an einer bestimmten Position auf dem Spielbrett steht, alle möglichen Nachfolgezustände ermittelt und diese in einer Liste abspeichert. Wird diese Funktion dann auf allen Feldern aufgerufen, auf denen eine Figur der Farbe steht, die gerade am Zug ist, so erhält man alle möglichen Nachfolgekonfigurationen des Spielbretts.

3.3 Der Minimax-Algorithmus

Da wir nun die Möglichkeit haben, die Menge der zulässigen Nachfolgezustände zu einem gegebenen Zustand zu ermitteln, können wir beginnen unter diesen Nachfolgezuständen nach dem „besten“ zu suchen. Wie bereits früher erwähnt, wird es voraussichtlich nicht immer möglich sein, optimale Nachfolger zu finden, da dafür meistens der Suchraum zu groß sein wird. Theoretisch könnte man jedoch optimale Nachfolger mit Hilfe des *Minimax*-Algorithmus [9] finden; dessen Funktionsweise soll hier kurz vorgestellt werden.

Der *Minimax* ist ein Algorithmus zur optimalen Lösung von *Zwei-Personen-Spielen*. Für unsere Zwecke genügt es, den Spezialfall für *Nullsummen-Spiele* [7] zu betrachten, da bei *Congo* jeder Nutzen für den einen Spieler, den gleichen Schaden für den anderen bedeutet. Wir werden im weiteren den weißen Spieler als *Max*-Spieler und den schwarzen als *Min*-Spieler auffassen, das heißt, wir betrachten den Nutzen immer aus der Sicht des weißen Spielers. Der Algorithmus betrachtet den vollständigen Suchraum, wobei mit steigender Tiefe der Knoten von Ebene zu Ebene, abwechselnd von *Max-Knoten* und von *Min-Knoten* gesprochen wird – je nach dem ob der maximierende Spieler oder der minimierende Spieler am Zug ist. Gehen wir also zum Beispiel einmal davon aus, dass gerade Weiß am Zug ist, dann wäre also der Wurzelknoten des Suchraums ein *Max-Knoten*, alle seine Kinder entsprechend *Min-Knoten*, usw.. Im Idealfall, das heißt, falls der gesamte Suchraum in realistischer Zeit und mit realisierbarem Platzbedarf durchsucht werden

könnte, würden wir einfach allen Endzuständen bei denen Weiß gewonnen hat den Wert `+1` und allen bei denen Schwarz gewonnen hat den Wert `-1` zuordnen, und wir hätten einen Algorithmus, der immer ein optimales Ergebnis liefert. Gehen wir davon aus, dass dieser Idealfall nicht eintrifft, so müssen wir für jeden Zustand eine Art *Bewertungsfunktion* einführen, die sinnvollerweise nur Werte zwischen `-1` und `+1` annehmen sollte.

Der Algorithmus besteht aus einer Tiefensuche. Er beginnt bei der Wurzel und wählt einen Pfad bis zu einem Blatt oder bis zu einem Knoten in der tiefsten Suchtiefe, die zuvor festgelegt wurde. Nun wird der Eltern-Knoten dieses Blattes (bzw. Knotens) betrachtet. Ist dieser Knoten ein *Max-Knoten*, so speichert der Algorithmus den maximalen Wert der Blätter (bzw. Nachfolgeknoten) dieses Knotens, andernfalls den minimalen. Hat ein Knoten K sowohl Blätter als auch weitere Knoten als Nachfolger, so müssen auch diese Knoten expandiert und ausgewertet werden um den korrekten Wert für den Knoten K bestimmen zu können. Entsprechend speichern *Max-Knoten* in höheren Ebenen jeweils den maximalen Wert ihrer Nachfolge-Knoten, während *Min-Knoten* den minimalen Wert ihrer Nachfolger speichern. Auf diese Weise wird simuliert, was ein Spieler optimalerweise an einer bestimmten Position im Spiel tun würde, wenn er davon ausgehen kann, dass sein Gegner danach auch optimal handelt. Als Ergebnis landet in der Wurzel des Suchraums das bestmögliche Ergebnis für den Spieler, der gerade am Zug ist. In Abb.4 ist ein kurzes Beispiel zur Veranschaulichung dargestellt.

```
int MiniMax(Knoten,Tiefe) {
    bisher_bester_Wert = -UNENDLICH
    if (Knoten ist ein Blatt oder maximale Suchtiefe ist erreicht) {
        return Bewertung(Knoten)
    }
    else {
        generiere alle möglichen Nachfolgezustände
        for (NFZ in Nachfolgezustände) {
            neuer_wert = -MiniMax(NFZ,Tiefe+1)
            bisher_bester_wert = Maximum(neuer_wert, bisher_bester_wert)
        }
    }
    return bisher_bester_wert
}
```

Abbildung 3: Minimax: Pseudo-Code

Ein gängiger Ansatz, einen solchen Algorithmus zu implementieren, besteht darin, jeweils eine Funktion für den Spieler *Min* und den Spieler *Max* zu schreiben, welche sich dann gegenseitig aufrufen, wie im Buch von Russell und

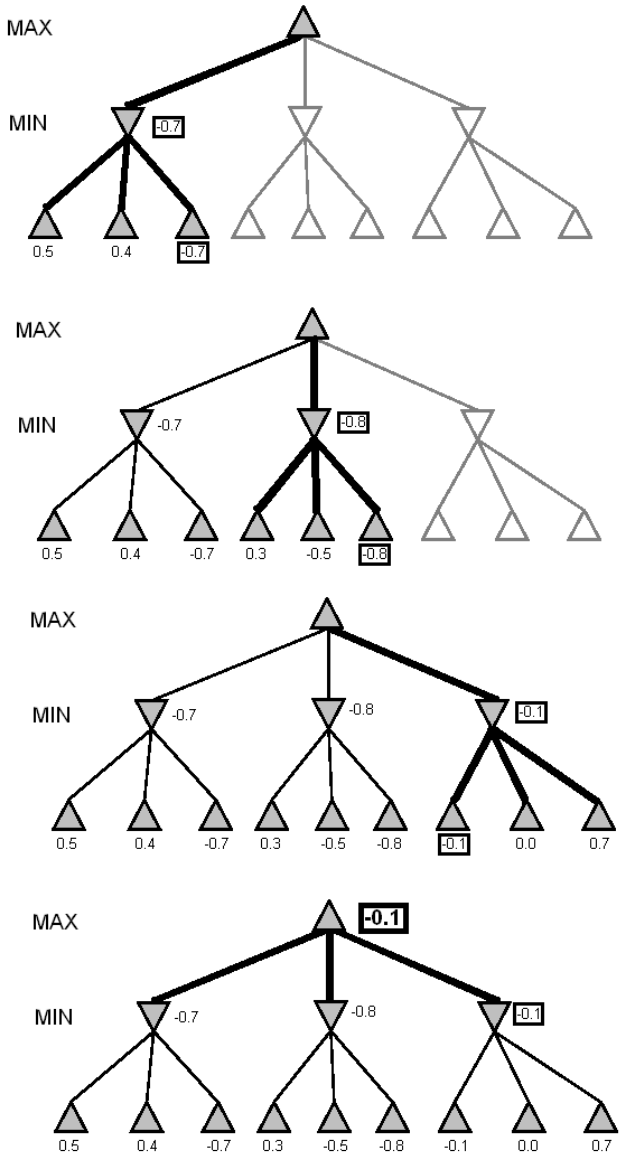

Abbildung 4: Minimax-Beispiel

Norvig beschrieben [9]. Es ist jedoch auch möglich auf diese Fallunterscheidung zu verzichten, indem man bei den rekursiven Aufrufen die Vorzeichen wechselt [6]. Dies hat den Vorteil, dass der Programmcode überschaubarer bleibt und leichter zu warten ist.

Die Variante des Minimax-Algorithmus aus Abb.3 liefert den Wert des aktuellen Knotens aus der Sicht des Spielers, der gerade am Zug ist, das heißt, je größer das Ergebnis, umso besser ist es für den aktuellen Spieler. Auf diese Weise kann auf überflüssigen Programmcode verzichtet und uns spätere Modifikationen am Code erleichtert werden. Zu beachten ist hierbei jedoch, dass beim initialen Aufruf der Funktion unterschieden werden muss, welcher der beiden Spieler gerade am Zug ist – ist nämlich der *Min-Spieler* am Zug, so muss das Endergebnis für unsere Zwecke noch negiert werden. Diese Variante des Algorithmus wird auch häufig als *Nega-Max* bezeichnet.

Diesen Algorithmus werden wir als Grundlage für alle weiteren Algorithmen verwenden. Der große Nachteil dieses Algorithmus ist, dass er den gesamten Suchraum (bis zur gegebenen maximalen Tiefe) durcharbeiten muss, was auf Grund des recht großen Verzweigungsgrads des Congo-Spiels von durchschnittlich 24 möglichen Nachfolgezuständen, leider dazu führt, dass der *Minimax*-Algorithmus bereits bei geringen Suchtiefen von sieben oder acht Halbzügen mehrere Stunden lang beschäftigt werden kann.

In Kapitel 4 wird gezeigt werden, wie man die Laufzeit des Algorithmus verbessern kann, ohne dabei an Qualität zu verlieren. In Kapitel 5 werden dann einige Bewertungsfunktionen vorgestellt, mit deren Hilfe die nichtterminalen Zustände in der maximalen Suchtiefe evaluiert werden können.

4 Der $\alpha\beta$-Algorithmus

4.1 Die Grundlage

Der $\alpha\beta$-Algorithmus ist eine optimierte Variante des *Minimax*-Algorithmus, der ebenfalls optimale Ergebnisse liefert, dafür jedoch nicht alle Knoten des Suchraums betrachten muss. Die grundlegende Idee ist, dass man auf Grund der bereits besuchten Knoten mit Sicherheit ausschließen kann, dass es in bestimmten Teilbäumen eine bessere Lösung geben kann, weshalb man diese dann erst gar nicht mehr betrachten muss. Dazu werden während der Suche zwei zusätzliche Werte α und β gespeichert, die ständig aktualisiert werden und dabei Informationen darüber beinhalten, welche Ergebnisse die Spieler bei perfekter Spielweise erzielen können.

Genauer spiegelt α das Ergebnis wieder, das der *Max*-Spieler mit Sicherheit erreichen kann, während β den Wert repräsentiert, den der *Min*-Spieler

niemals überschreiten muss. Besitzt ein *Max*-Knoten ein Kind, dessen α-Wert den β-Wert übersteigt, so wäre es für den *Min*-Spieler offensichtlich nicht sinnvoll diesen *Max*-Knoten auszuwählen – d.h. er würde ihn nie auswählen – weshalb es an dieser Stelle nicht notwendig ist, die weiteren Kinder dieses Knotens zu betrachten, da er in der Praxis sowieso nie erreicht werden würde. Die weitere Suche in diesem Teilbaum kann also abgebrochen werden, und man spricht von einem *β-cutoff*. Genau umgekehrt sieht es dann bei *Min*-Knoten aus – hier spricht man entsprechend von *α-cutoffs*. In Abb.5 ist der Code für einen $\alpha\beta$-Algorithmus skizziert, der sich leicht aus der Minimax-Variante aus Abb.3 gewinnen lässt.

```
int AlphaBeta(Knoten,Tiefe,Alpha,Beta) {
    bisher_bester_Wert = -UNENDLICH
    if (Knoten ist ein Blatt oder maximale Suchtiefe ist erreicht) {
        return Bewertung(Knoten)
    }
    else {
        generiere alle möglichen Nachfolgezustände
        for (NFZ in Nachfolgezustände) {
            neuer_wert = -AlphaBeta(NFZ,Tiefe+1,-Beta,-Alpha)
            if (neuer_wert >= Beta)
                return Beta
            Alpha = Maximum(neuer_wert, Alpha)
        }
    }
    return Alpha
}
```

Abbildung 5: $\alpha\beta$: Pseudo-Code

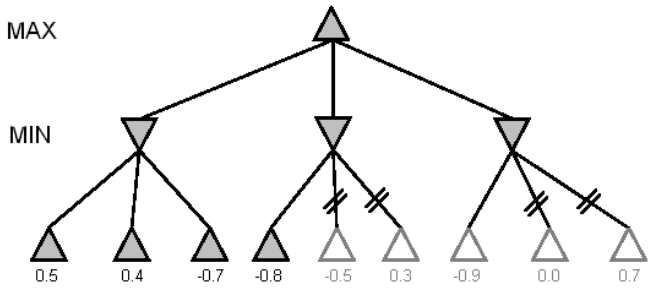

Abbildung 6: Expandierte Knoten mit $\alpha\beta$ von links nach rechts ausgeführt

In dem Beispiel aus Abb.6 müssen nur fünf von neun Blättern betrachtet werden, was also im Vergleich zum *Minimax*-Algorithmus eine wesentliche Beschleunigung ergibt. In Bäumen mit größerer Tiefe sind nicht nur Blätter von *cutoffs* betroffen, sondern ganze Teilbäume, was selbstverständlich noch zu einem deutlich größeren Zeitgewinn bei der Suche führt. Wie anhand des Beispiels schon vermutet werden kann, spielt die Reihenfolge, in der die einzelnen Knoten betrachtet werden, eine sehr große Rolle für die Effizienz des $\alpha\beta$-Algorithmus.

4.2 Zuganordnung

Betrachtet man z.B. einen Suchraum, der fast identisch zu dem Suchraum aus Abb.6 ist, bei dem jedoch die Reihenfolge der Blätter in den beiden rechten Teilbäumen verdreht ist (vgl. Abb.7), so erkennt man, dass hier keine *cutoffs* möglich sind, falls man die Blätter wieder der Reihe nach von rechts nach links betrachtet, weil das relevante Blatt jedes der beiden Teilbäume, das einen *cutoff* verursachen würde, jeweils als letztes betrachtet wird.

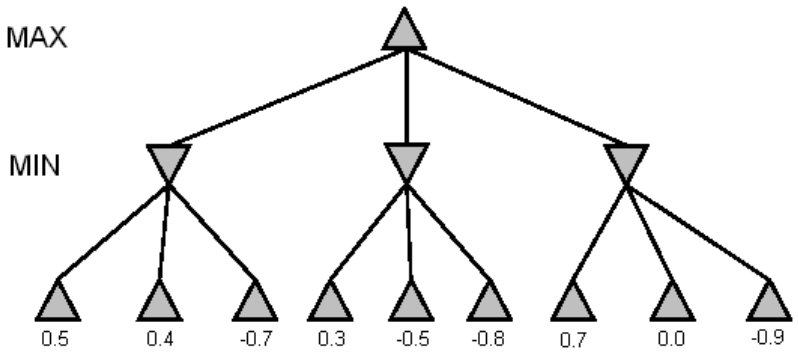

Abbildung 7: $\alpha\beta$-Beispiel mit anderer Reihenfolge der Blätter – hier sind keine *cutoffs* möglich, wenn der Suchraum von links nach rechts traversiert wird

Bisher können wir festhalten, dass uns die $\alpha\beta$-Suche gegenüber dem *Minimax*-Algorithmus nur Vorteile bringt, da sie nur solche Knoten nicht betrachtet, die für eine optimale Entscheidung vollkommen irrelevant sind. Mittels $\alpha\beta$ werden wir also das gleiche Ergebnis erhalten wie mit *Minimax* – nur in kürzerer Zeit.

Wie wir bereits gesehen haben spielt die Reihenfolge, in der die Knoten eines Suchraums von einem $\alpha\beta$-Algorithmus betrachtet werden, eine sehr große

Rolle. Im schlechtesten Fall kann es passieren, dass gar keine *cutoffs* möglich sind und wir den ganzen Suchraum untersuchen müssen. Um dem entgegenzuwirken, versucht man die Knoten, die möglicherweise einen *cutoff* erzeugen könnten, zuerst zu betrachten. Bei solch einem Verfahren spricht man von *Zuganordnung* (engl. move-ordering) – man bringt die Knoten in eine vorteilhafte Reihenfolge. Wie man vielleicht schon anhand der vorangegangenen Beispiele erahnen kann, sind vor allem „extreme“ Knoten solche, die *cutoffs* erzwingen können. Das heißt, irgendwelche Aktionen, die auf dem Spielbrett nur wenig bewirken – die Bewertung des Zustandes also kaum verändern – sollten nicht unbedingt zuerst betrachtet werden, da sie wahrscheinlich keinen *cutoff* bewirken werden. Eine einfache Methode um die Wahrscheinlichkeit von *cutoffs* zu erhöhen besteht darin, zuerst solche Zustände zu betrachten, bei denen Figuren geschlagen wurden, da das in der Regel[2] zu einer Änderung der Bewertung des Spielzustands führt.

Unser $\alpha\beta$-Algorithmus wird also zunächst dahingehend modifiziert werden, dass zunächst solche Nachfolgezustände betrachtet werden, bei denen Figuren geschlagen werden. Eine weitere Modifikation, die wir vornehmen werden ist, offensive Züge vor defensiven zu betrachten. Dies führte bei zahlreichen Testläufen dazu, dass sich die Anzahl der *cutoffs* deutlich erhöht hat, was vermutlich daher kommt, dass bei offensiven Zügen die Wahrscheinlichkeit steigt, dass sich der Wert der Bewertungsfunktion in den folgenden Zügen ändert. Um eine möglichst gute *Zuganordnung* zu erhalten, sind einige Versuche notwendig, die dann miteinander verglichen werden können, da wir uns natürlich nicht für jede einzelne Situation im Voraus überlegen wollen, welche Aktion tatsächlich einen *cutoff* verursacht. Hier müssen wir uns also leider mittels empirischen Ergebnissen für eine geeignet scheinende *Zuganordnung* entscheiden. Da diese ganzen Modifikationen unmittelbar die `move`-Funktion betreffen, muss für sie direkt in den bereits bestehenden Code eingegriffen werden, was die ganze Sache etwas aufwendig gestaltet. Wie wir in Kapitel 6 sehen werden, ist die Verbesserung, die durch eine geschickte *Zuganordnung* erreicht werden kann, die Mühen jedoch unbedingt wert.

4.3 Transpositionstabelle

Es ist de facto so, dass in dem Suchraum, den unser $\alpha\beta$-Algorithmus durchläuft, sehr viele Zustände mehrfach vorkommen und jedesmal aufs Neue exploriert werden müssen. Hier ist ein weiterer Ansatzpunkt um den Algorithmus zu verbessern. Man kann eine sogenannte *Transpositionstabelle* anlegen,

[2] *In der Regel* bedeutet bei Verwendung der meisten Bewertungsfunktionen, wie z.B. denen aus Abschnitt 5.1 oder Abschnitt 5.4 – es gibt jedoch auch Bewertungsfunktionen, bei denen dies nicht zwangsläufig zutreffen muss, vgl. Abschnitt 5.2

die dazu dient, bereits explorierte Zustände – zusammen mit dem Ergebnis der Exploration – zu speichern. Trifft man an einer anderen Stelle im Suchraum auf den selben Zustand noch einmal, so kann man in der Transpositionstabelle nachsehen, was früher für diesen Zustand für ein Ergebnis erzielt wurde, und unter Umständen auf die nochmalige Exploration dieses Knotens im Baum verzichten. Immer wird das leider nicht möglich sein, da wir berücksichtigen müssen, an welcher Stelle im Baum (d.h. in welcher Tiefe) sich der Knoten befindet, bzw. an welcher Stelle sich der Knoten befand, der bereits früher ausgewertet wurde. Haben wir zum Beispiel in der Transpositionstabelle einen Zustand, der sich in der tiefsten Suchebene befand, ausgewertet, und treffen wir im weiteren Verlauf der Suche auf den gleichen Zustand nochmal, jedoch noch zwei „Tiefenstufen“ von der tiefsten Ebene entfernt, so wird uns der Wert aus der Transpositionstabelle nicht zufriedenstellen.

Dies lässt sich sehr schön an einem Beispiel nachvollziehen: In Abb.8 soll der Minimax-Wert für y bestimmt werden. Es ist gut zu erkennen, dass die Information, die uns aus dem ersten Untersuchen von Zustand x vorliegt, nicht ausreicht, um eine vergleichbare Qualität beim zweiten Mal zu erhalten, da sich beim ersten Mal der Wert für x aus der direkten Evaluierung des Zustandes ergab, während sich beim zweiten Mal die Bewertung von Zustand x aus den Bewertungen der Nachfolger ergeben muss, was natürlich zu einem deutlich aussagekräftigeren Wert führt. Was jedoch genau im rechten Teilbaum passiert, wissen wir erst, wenn wir diesen exploriert haben.

Wir müssen also sehr vorsichtig sein, wann wir Informationen aus der Transpositionstabelle verwenden dürfen, ohne dass dadurch Qualität verloren geht. Das heißt, wir müssen zusammen mit dem Zustand auch immer die Tiefe im Baum speichern, in der sich der Zustand befindet. Ferner muss natürlich auch jedesmal ein Wert für den Zustand gespeichert werden. Hier stellt sich jedoch die Frage was für einen Wert man geschickterweise speichern soll, da man bei der $\alpha\beta$-Suche meistens keinen exakten Wert für einen Knoten erhält, sondern meist nur eine obere oder eine untere Schranke, gegeben durch den β- oder den α-Wert. Um zu wissen, um welchen Wert es sich bei dem in der Transpositionstabelle gespeicherten Wert tatsächlich handelt, ist es also erforderlich, diese Information auch noch zu speichern. Dies tun wir mit Hilfe eines *Flags*, das uns anzeigt, ob es sich bei dem Wert um einen exakten Wert, einen β-Wert oder einen α-Wert handelt. Finden wir nun einen Zustand in der Transpositionstabelle, dann können wir mittels des Flags entscheiden, wie weiter vorzugehen ist. Für das genaue Vorgehen vergleiche Abb.9:

Die Funktion `probeHash` testet, ob ein Zustand bereits gespeichert ist und vergleicht dann die Parameter des gespeicherten Zustands mit denen des aktuellen Zustands um danach zu entscheiden, ob darauf verzichtet werden

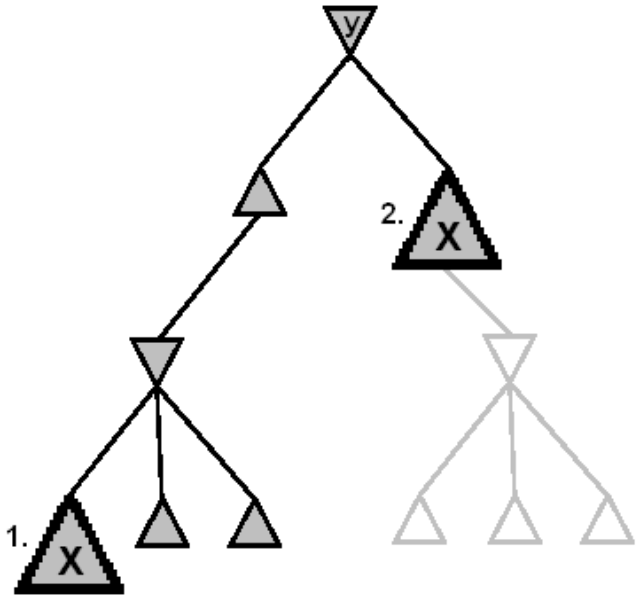

Abbildung 8: Gleiche Zustände, jedoch andere Suchtiefe

kann, den aktuellen Zustand weiter zu untersuchen oder nicht.

Um eine Transpositionstabelle möglichst effizient zu gestalten, verwenden wir dafür eine *Hashtabelle* [8]. Hashtabellen verfügen über sehr gute Zugriffszeiten, sowohl beim Speichern als auch beim Auslesen von Werten. Da wir idealerweise alle Zustände, die wir explorieren, auch speichern wollen, bietet es sich an, dynamische Hashverfahren zu verwenden, das heißt solche, bei denen die Größe der Hashtabelle nicht von Anfang an fest vorgegeben ist, sondern bei Bedarf noch größer werden kann. Für einen bestimmten Zustand wird ein bestimmtes Fach in der Hashtabelle errechnet, in dem er gespeichert wird. Da es hierbei auch dazu kommen kann, dass mehrere Zustände in dem selben Fach abgelegt werden, kann die Hashtabelle unter Umständen sehr groß werden, was in der Praxis leider sehr schnell dazu führen kann, dass der Arbeitsspeicher eines Rechners überlastet wird, weswegen wir uns – zusätzlich zu den dynamischen Hashtabellen – auch mit Tabellen konstanter Größe beschäftigen werden. Dabei kann es dann allerdings passieren, dass Zustände, die in der Tabelle gespeichert sind, von anderen überschrieben werden und entsprechend nicht mehr aus der Hashtabelle abrufbar sind. Welches der beiden Verfahren sich im Endeffekt als geeigneter herausstellen wird, werden wir

```
int probeHash(int depth, int alpha, int beta, State actualState) {
   if (hash->find(actualState)) {
      if (savedState.depth >= depth) {
         if (savedState.flag == flagEXACT)
            return savedState.value;
         if ((savedState.flag == flagALPHA) && (savedState.value <= alpha))
            return alpha;
         if ((savedState.flag == flagBETA) && (savedState.value >= beta))
            return beta;
      }
   }
   return value_UNKNOWN;
}
```

Abbildung 9: Verfahren, das angibt, wie mit unterschiedlichen Flags umzugehen ist

in Kapitel 6 sehen. Es wird sich jedenfalls zeigen, dass durch das Verwenden von Transpositionstabellen egal welcher Art, einiges an Rechenzeit gewinnen lässt.

5 Die Bewertungsfunktion

Die Bewertungsfunktion (oder Evaluierungsfunktion) stellt einen sehr wichtigen Teil bei der Implementierung eines Agenten für ein Spiel wie *Congo* dar. Gäbe es eine Bewertungsfunktion, die den tatsächlichen spieltheoretischen Wert eines Zustandes widerspiegeln würde, dann könnte man einfach immer denjenigen Nachfolgezustand wählen, für den die Bewertungsfunktion den höchsten Wert ausgibt und schon hätte man einen perfekten Algorithmus. So eine perfekte Evaluierungsfunktion gibt es jedoch in den meisten Fällen nicht, bzw. sie bräuchte viel zu viel Rechenzeit. Doch trotz dieses Wissens ist eine Bewertungsfunktion bei *Congo* unerlässlich, da der Suchraum zu groß ist, um komplett durchsucht zu werden und wir gezwungen sind, in einer gewissen Tiefe die Suche abzubrechen. Genau an dieser Stelle braucht man entsprechend eine Bewertungsfunktion, die einen ungefähren Wert des Zustands berechnet, bei dem die Suche unterbrochen wurde. Ohne eine Bewertung dieses Zustands wäre es unmöglich, Algorithmen wie *Minimax* oder $\alpha\beta$ auszuführen, da die Rekursionsbasis komplett fehlen würde. Statt der tatsächlichen spieltheoretischen Werte, werden entsprechend die geschätzten Werte der Bewertungsfunktion genommen, um eine Rekursionsbasis zu erhalten.

5.1 Eine einfache Bewertungsfunktion

Bei schachähnlichen Spielen ist es recht gebräuchlich – und auch vergleichsweise einfach – den aktuellen Zustand anhand der sich noch auf dem Spielbrett befindenden Figuren zu bewerten. Aus diesem Grund, wollen wir zunächst eine recht einfache Bewertungsfunktion implementieren, die nur darauf basiert, wie viele Spielfiguren jeder Spieler noch auf dem Brett hat.

Bei *Schach* gibt es sehr große Unterschiede in der Stärke der einzelnen Spielfiguren. Die Dame ist zum Beispiel sehr mächtig, während der Bauer vergleichsweise kaum von Bedeutung ist. Eine bei *Schach* gebräuchliche Faustregel ist, dass zum Beispiel ein Turm fünf mal soviel wert ist wie ein Bauer, und dass eine Dame den neunfachen Wert eines Bauern hat. Das sind sogenannte *Tauschwerte*, die sich aus viel Erfahrung und langjährigem Spiel ergeben haben, die sich für *Congo* leider nicht in der Literatur finden lassen.

Es ist jedoch recht einfach zu erkennen, dass es bei *Congo* keine so stark ausgeprägten Unterschiede in der Mächtigkeit der Figuren gibt – sie sind alle ziemlich ausgeglichen und es ist nicht so deutlich ersichtlich, welche Figuren anderen gegenüber besser oder schlechter sind. Dies lässt sich schon daran festmachen, dass die Diskrepanz in der Anzahl der Zugmöglichkeiten der verschiedenen Figuren, im Vergleich zu *Schach*, ziemlich gering ist.

Auf Grund dieser Erkenntnis werden wir zunächst eine ganz einfache Bewertungsfunktion konstruieren, die wir dadurch erhalten, dass wir annehmen, dass tatsächlich alle Figuren *gleichmächtig* sind. Wir geben jeder Figur den Tauschwert 1, außer dem Löwen, der unendlich wertvoller ist, da sein Verlust automatisch zu einer Niederlage führt. Ihm geben wir den Wert 1000. Um daraus nun eine Bewertungsfunktion zu erhalten, vereinbaren wir einfach, dass die weißen Figuren positive und die schwarzen negative Werte erhalten. Dann müssen wir einfach nur noch die Werte aller Figuren auf dem Brett addieren und wir erhalten eine, für den Anfang ausreichende, Bewertungsfunktion. Um das Ganze mit den Aussagen aus Abschnitt 3.3 bezüglich der Gewinngrenzen in Einklang zu bringen, müssen wir noch ein wenig umskalieren. Dazu können wir aber einfach die Grenze für den Sieg statt auf ± 1, zum Beispiel auf ± 900 festlegen. Wie man sich leicht ausrechnen kann, ist damit sichergestellt, dass jede Sieg-Situation erkannt wird und auch keine andere fälschlicherweise als solche erkannt werden kann.

5.2 Monte-Carlo-Evaluation

In diesem Abschnitt wird eine andere Form der Evaluationsfunktion vorgestellt, wie sie Abramson beschrieben hat [1]. Die Idee dabei ist, vollkommen weg von den Tauschwerten der einzelnen Figuren zu gehen und statt

dessen zu schauen, wie sich das Spiel von einem bestimmten Zustand aus weiter entwickeln könnte. Hierzu werden, von dem zu evaluierenden Zustand aus, mögliche Nachfolger-Sequenzen zufällig generiert und dahingehend überprüft, ob die eine oder die andere Partei am Ende gewonnen hat. Bei dieser Art von Methoden spricht man von *Monte-Carlo-Simulation* [2].

Genauer gesagt sucht man sich, von dem aktuellen Zustand ausgehend, zufällig einen Nachfolger aus und wiederholt dies dann iterativ von dem Nachfolger ausgehend. In einer bestimmten Tiefe T bricht man ab, falls man bis dort noch keinen Endzustand erreicht hat – wird ein solcher schon zuvor erreicht, wird sofort abgebrochen. Diese ganze Prozedur wiederholt man n mal, wobei n zu Beginn festgelegt wurde. Die Differenz von gewonnenen und verlorenen Spielen (aus der Sicht eines der beiden Spieler), ergibt dann die Bewertung des Zustandes. Je größer man n wählt, das heißt, je öfter man diese Prozedur wiederholt, desto besser wird die Bewertungsfunktion. Auch eine Erhöhung von T hat eine Verbesserung der Evaluation zur Folge – für ein kleines T ist es sehr wahrscheinlich, dass sehr viele Zustände mit 0 bewertet werden. Geht man jedoch tiefer, so steigt die Wahrscheinlichkeit einen Endzustand zu erreichen, was eine Verbesserung der Bewertungsfunktion bewirkt.

Je größer jedoch auf der anderen Seite die Werte für n und T gewählt werden, desto aufwändiger wird die Evaluation eines Zustandes und desto länger wird sie dauern. Insbesondere für große Suchtiefen steht also zu befürchten, dass diese Evaluierungsmethode zu zeitintensiv sein könnte. Diese Vermutung hat sich bei zahlreichen Testläufen leider bewahrheitet (vgl. Abschnitt 6.2).

5.3 Randomisierte Bewertungsfunktion

Noch eine andere Möglichkeit der Bewertungsfunktion stellt die *Randomisierte Bewertungsfunktion* dar. Für jeden Zustand in der maximalen Suchtiefe wird einfach eine zufällige Zahl generiert, die diesem dann als Bewertung zugewiesen wird.

Durch eine zufällige Evaluationsfunktion werden solche Zustände bevorzugt, die mehr Nachfolgezustände besitzen als andere. Nehmen wir an K sei ein beliebiger Max-Knoten in unserem Suchraum. Entsprechend ergibt sich der Nutzen für K als das Maximum aller Nutzenwerte seiner Nachfolgeknoten. Der Erwartungswert für den Nutzen für den Zustand K wächst also bei einer zufälligen Evaluationsfunktion direkt mit der Anzahl der Nachfolger.

Auf diese Art und Weise führt eine zufällige Evaluierungsfunktion dazu, dass solche Knoten bevorzugt werden, die viele Nachfolger besitzen. Zustände mit vielen Nachfolgern sind jedoch flexibel, das heißt, ein Spieler hat viele

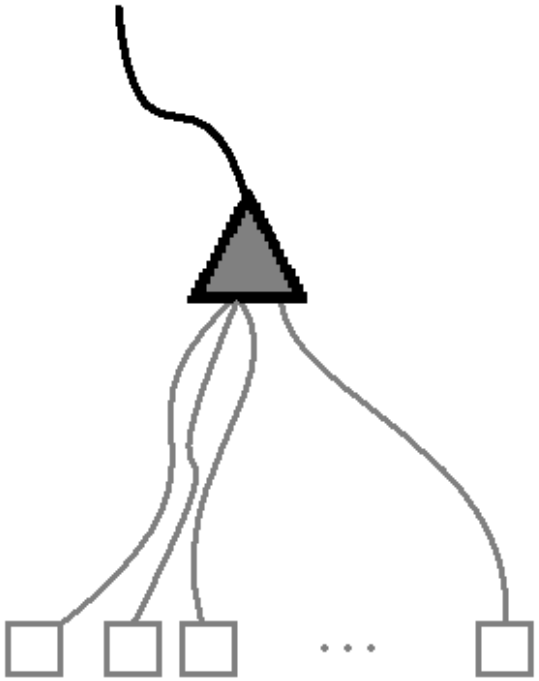

Abbildung 10: Monte-Carlo-Evaluation

Optionen zu handeln und wird nicht so leicht zu einer bestimmten Aktion gedrängt. Besonders zu Beginn der $\alpha\beta$-Suche, wenn es noch sehr viele Zugmöglichkeiten gibt, könnte dieses Verfahren vernünftige Ergebnisse erwarten lassen. Sind jedoch nur noch wenige Figuren auf dem Spielbrett und das Ende bereits recht nahe, so ist zu vermuten, dass andere Verfahren bessere Resultate liefern werden.

5.4 Gewichtung von Figuren

In Abschnitt 5.1 haben wir eine sehr einfache Bewertungsfunktion kennen gelernt, die alle Figuren außer dem Löwen gleich gewichtet. Die Vermutung liegt nahe, dass diese Art der Bewertung jedoch nicht sehr exakt die Qualität eines Zustands widerspiegeln wird. Ist man in der Lage genügend tief zu suchen, das heißt, trifft man häufig auf einen Endzustand, oder ist die Suche zumindest so tief, dass sich häufig klare Tendenzen erkennen lassen, wer voraussichtlich gewinnen wird (z.B. wenn ein Spieler doppelt so viele Figuren hat als der andere), dann ist diese einfache Bewertungsfunktion wahrscheinlich ausreichend, da sie zwischen solchen Zuständen ausreichend

gut unterscheiden kann. In der Praxis sieht es aber eher so aus, dass sich die Zustände in der maximalen Suchtiefe oftmals nur geringfügig unterscheiden und diese einfache Evaluierungsfunktion nur recht spärlich zwischen ihnen differenzieren kann. Aus diesem Grund wollen wir versuchen, diese einfache Bewertungsfunktion zu verbessern.

Wie bereits erwähnt, werden zum Beispiel bei *Schach* den unterschiedlichen Figuren auch unterschiedliche *Tauschwerte* zugeordnet. Genau den gleichen Ansatz wählen wir nun auch bei *Congo*. Da hierfür leider nur unzureichend Literatur existiert, möchte ich in Tabelle 1 einmal eine Tauschwertetabelle angeben, die mir auf Grund eigener Beobachtungen als sinnvoll erscheint:

Figur	Tauschwert
Bauer	1
Zebra	1,5
Giraffe	2
Elefant	2
Affe	2
Krokodil	2
Superbauer	2
Löwe	1000

Tabelle 1: Mögliche Tauschwerte von Figuren

Wie bei *Schach* wurde der Bauer als Grundwert gewählt. Im Verhältnis zu den anderen Figuren, ist er hier jedoch schon recht wertvoll, da er deutlich mehr Zugmöglichkeiten hat als ein Schach-Bauer. Die meisten anderen Figuren scheinen mir relativ gleichwertig zu sein, mit Ausnahme des Zebras, dass sich in zahlreichen Simulationen als etwas schwächer herausstellte. Diese Zahlenwerte spiegeln natürlich mit Sicherheit noch nicht exakt die Werte der Figuren im Spiel wider, sie haben sich jedoch in zahlreichen Versuchen als recht vernünftig erwiesen.

Diese Bewertungsfunktion stellt keine besonders große Neuerung dar – es wird lediglich zwischen den einzelnen Figuren differenziert. Der große Vorteil ist auch hier die Einfachheit der Funktion. Sie ist sehr schnell zu berechnen, was für uns von wesentlicher Bedeutung ist, da diese Funktion beim Lauf unseres Algorithmus voraussichtlich sehr häufig aufgerufen werden muss – nämlich jedesmal, wenn der Algorithmus die tiefste Suchebene erreicht. Wie oft das in etwa der Fall ist, kann Tabelle 2 entnommen werden.

Suchtiefe	Minimax	$\alpha\beta$
4	$\approx 3,3 \cdot 10^5$	≈ 4000
8	$\approx 1,1 \cdot 10^{11}$	$\approx 1,7 \cdot 10^6$
12	$\approx 3,6 \cdot 10^{16}$	$\approx 6,8 \cdot 10^{10}$
16	$\approx 1,2 \cdot 10^{22}$	$\approx 2,8 \cdot 10^{14}$

Verzweigungsfaktor Minimax ≈ 24
Verzweigungsfaktor $\alpha\beta$-Suche ≈ 8

Tabelle 2: Durchschnittliche Anzahl der Bewertungsfunktions-Aufrufe bei 20 verschiedenen, zufällig gewählten Ausgangszuständen für die Suche

6 Ergebnisse

6.1 Durchschnittliche Laufzeiten

Um nun die verschiedenen Algorithmen, bzw. Algorithmenerweiterungen, die vorgestellt wurden, gegeneinander abwägen zu können, haben wir in Tabelle 3 die Laufzeiten einiger Algorithmen bei verschiedenen Suchtiefen aufgelistet. Die angegebenen Laufzeiten ergaben sich als Mittel aus der zufälligen Auswahl von zehn unterschiedlichen Ausgangszuständen, wobei als Evaluierungsfunktion jedesmal die Funktion aus Abschnitt 5.4 diente, die mit Hilfe der *gewichteten Tauschwerte* arbeitet. Dabei muss erwähnt werden, dass alle angegebenen Algorithmen optimal sind, das heißt, insbesondere liefern sie bei gleicher Suchtiefe auch alle das selbe Resultat.

Wie man Tabelle 3 sehr gut entnehmen kann, liefert der $\alpha\beta$-Algorithmus im Vergleich zum *Minimax* deutlich bessere Laufzeiten – eine sehr starke Beschleunigung ist offensichtlich mittels verbesserter Zuganordnung erreicht worden, was genau dem entspricht, was wir schon in der Theorie vermutet hatten (vgl. Abschnitt 4.2).

Mittels *Hashing* konnte die benötigte Laufzeit im Schnitt nochmal etwas mehr als halbiert werden. Dabei zeigte sich, dass kein großer Unterschied in den Laufzeiten zwischen dynamischen und konstanten Hashverfahren zu erkennen ist. Als Hashtabelle für das konstante Verfahren wurde eine Tabelle mit etwas mehr als 1,5 Millionen Feldern gewählt, was sich in mehreren Testläufen als ein guter Wert herausstellte. Ab einer bestimmten Tiefe führte das dynamische Hashing dazu, dass der Arbeitsspeicher restlos ausgelastet wurde und das Programm abstürzte, weshalb für unser Programm tendenziell das Hashverfahren mit einer Hashtabelle von konstanter Größe vorzuziehen

Such-tiefe	Minimax	$\alpha\beta$ ohne Zug-anordnung	$\alpha\beta$ mit Zug-anordnung	$\alpha\beta$ mit dynam.Hash	$\alpha\beta$ mit konst.Hash
3	0s	0s	0s	0s	0s
4	1s	0s	0s	0s	0s
5	30s	2s	0s	0s	0s
6	1.200s	30s	0s	0s	0s
7	40.000s	1.050s	3s	2s	2s
8	–[1]	12.500s	15s	9s	7s
9	–[1]	–[1]	90s	50s	50s
10	–[1]	–[1]	440s	180s	185s
11	–[1]	–[1]	2.500s	–[2]	1.100s
12	–[1]	–[1]	12.500s	–[2]	5.100s

[1]) wegen Zeitüberschreitung nicht vorhanden
[2]) wegen Speicherüberschreitung nicht vorhanden

Tabelle 3: Durchschnittliche Laufzeiten der Algorithmen auf einem Intel Xeon 3.06GHz-Rechner mit 4GB Arbeitsspeicher

ist. Es benötigt vergleichbare Laufzeit, ist jedoch zuverlässiger, da es für alle Suchtiefen gleichermaßen funktioniert.

6.2 Durchschnittliche Laufzeiten bei anderen Evaluierungsfunktionen

Bisher haben wir uns in diesem Kapitel ausschließlich mit der Bewertungsfunktion aus Abschnitt 5.4 beschäftigt. Nun wollen wir noch die Laufzeiten bei Verwendung anderer Evaluationsfunktionen betrachten. In Tabelle 4 sind einige durchschnittliche Laufzeiten angegeben, die sich bei der Verwendung des $\alpha\beta$-Algorithmus mit konstanter Hashgröße ergaben.

Bei der *Monte-Carlo*-Evaluation wurden dabei jeweils 10 zufällige Pfade der Tiefe 10 gewählt. Bei größerer Tiefe, bzw. mehr Pfaden, waren die Laufzeiten entsprechend noch deutlich größer. Die Laufzeiten des $\alpha\beta$-Algorithmus mit der einfachen Bewertungsfunktion stellten sich zwar als schneller heraus als die mit der gewichteten Bewertungsfunktion, jedoch bewegten sich beide in der selben Größenordnung.

6.3 Computer vs. Computer

Mit Hilfe der Laufzeiten aus Tabelle 4 lässt sich jedoch nur eine Aussage darüber treffen, wie schnell die einzelnen Bewertungsfunktionen sind. Viel interessanter für uns ist jedoch die Frage, wie gut sie sind, das heißt, wie

Suchtiefe	gewichtete Tauschwerte	einfache Bewertung	Monte-Carlo-Evaluation	Randomisierte Evaluation
4	0s	0s	22s	0s
5	0s	0s	392s	0s
6	0s	0s	1700s	8s
7	2s	1s	–	21s
8	7s	5s	–	110s
9	50s	37s	–	900s
10	185s	120s	–	4600s
11	1.100s	600s	–	20000s
12	5.100s	2500s	–	–

Tabelle 4: Laufzeiten bei Verwendung verschiedener Evaluationsfunktionen

genau sie den tatsächlichen Nutzen eines bestimmten Zustands für einen Spieler widerspiegeln.

Um dies zu erreichen, werden wir einige Spiele zwischen zwei Spielern simulieren, die mit unterschiedlichen Bewertungsfunktionen arbeiten. Dazu lassen wir einfach zwei Agenten gegeneinander antreten, die Zustände mit unterschiedlichen Evaluationsfunktionen bewerten, und beobachten die Spielausgänge.

	gewichtet	einfach	Monte-Carlo	randomisiert
gewichtet	–	100:0	100:0	99:1
einfach		–	92:8	96:4
Monte-Carlo			–	69:31
randomisiert				–

Tabelle 5: Algorithmen im direkten Vergleich

In Tabelle 5 sind die Ergebnisse von jeweils 100 Simulationsläufen bei verschiedenen Suchtiefen zu sehen. Es fällt auf, dass offensichtlich die Evaluationsmethode der *gewichteten Tauschwerte* die beste der hier aufgeführten Methoden ist, da sie in fast allen Fällen zum Sieg führte. Die *Monte-Carlo*-Evaluationsmethode und die Methode der *randomisierten* Evaluation konnten leider nicht überzeugen und haben selbst im direkten Vergleich gegen die einfache Methode aus Abschnitt 5.1 sehr schlecht abgeschnitten.

7 Weitere Möglichkeiten

Der hier vorgestellte $\alpha\beta$-Algorithmus ist eine recht nahe liegende Methode zur Implementierung eines Agenten für ein Spiel wie *Congo*. Es gäbe jedoch auch noch einige andere Methoden die alternativ oder in Kombination angewendet werden könnten, zum Beispiel Methoden aus dem Bereich des *Rein-*

forcement Learnings, wie sie etwa im Buch von Sutton und Barto beschrieben werden [11]. Dabei handelt es sich um Algorithmen, die ihre Züge selbst – in Abhängigkeit vom erzielten Endergebnis – evaluieren und daraus lernen. Solche Methoden könnten auch gezielt für die Evaluierung von Zuständen eingesetzt werden. Andererseits gibt es jedoch auch noch zahlreiche Möglichkeiten unseren $\alpha\beta$-Algorithmus zu verbessern, ohne gleich ganz von vorne anzufangen. Im Folgenden werden einige Ideen vorgestellt, mit denen sich die Effizienz oder die Güte der $\alpha\beta$-Suche verbessern ließe.

Um die vergleichsweise zeitintensive Suche nach guten Nachfolgern in der Anfangsphase des Spiels zu verkürzen, könnte man eine *Eröffnungsbibliothek* anlegen, wie das zum Beispiel bei guten Schachprogrammen standardmäßig der Fall ist. Am Anfang ist der Brettzustand immer der gleiche und die Änderungen desselben in der Anfangsphase des Spiels sind überschaubar, so dass sich hier mit relativ wenig Speicherbedarf eine sinnvolle Zugbibliothek erstellen ließe. Da sich besonders zu Beginn des Spiels die $\alpha\beta$-Suche als sehr zeitintensiv erweist, könnte dadurch der durchschnittliche Zeitaufwand enorm reduziert werden.

Ein Ansatzpunkt zur Steigerung der Qualität des Programms, wäre die Bewertungsfunktion. Hier könnte man einerseits einfach versuchen die Tauschwerte der Figuren exakter zu bestimmen, indem man zahlreiche Simulationen mit verschiedenen Tauschwerten durchführt und sich schlussendlich für diejenigen entscheidet, welche die besten Ergebnisse liefern. Andererseits könnte man auch versuchen, bei der Bewertung eines Zustands Stellungsvorteile zu berücksichtigen. Zum Beispiel könnte man davon ausgehen, dass ein Krokodil, das sich im Fluss befindet, einen höheren Wert hat als eines, das sich außerhalb des Flusses befindet, da sich durch den Fluss die Bewegungsmöglichkeiten des Krokodils vergrößern. Befindet sich dagegen eine andere Figur im Fluss, so wird dies den Wert des Zustandes eher verringern, weil der Spieler durch sie leichter unter Druck gesetzt werden kann, da er ja die Figur im nächsten Zug wieder bewegen muss, wenn er sie nicht verlieren will. Ein anderes Beispiel wäre ein Bauer, der als einzige Figur zwischen zwei Löwen steht. Er ist in seiner Bewegungsfreiheit stark eingeschränkt, andererseits kann er jedoch nicht vom gegnerischen Löwen geschlagen werden ohne dass dieser sich selbst quasi „matt“ setzt. Solch ein Bauer könnte also unter Umständen auch einen anderen Tauschwert haben. Solche oder auch noch deutlich komplizierter Stellungsaspekte könnten mit in die Bewertungsfunktion einfließen und diese verbessern. Es muss jedoch aufgepasst werden, dass durch die Verbesserung der Bewertungsfunktion nicht der nötige Zeitaufwand zu groß wird, da wir sonst die Suchtiefe des Algorithmus wieder reduzieren müssten. An dieser Stelle den besten Mittelweg zu finden, könnte bei der Suche nach sehr guten Algorithmen, die wohl größte Herausforderung darstellen.

Literatur

[1] Abramson, Bruce: *Expected-Outcome: A General Model of Static Evaluation.* IEEE Transaction on pattern analysis and machine intelligence, 12(2), 1990.

[2] Cazenave, Tristan und Bernard Helmstetter: *Combining Tactical Search and Monte-Carlo in the Game of Go.* In: *Proceedings of IEEE Symposium on Computational Intelligence and Games*, Seiten 171–175, 2005.

[3] Coulom, Rémi: *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search.* In: *5th International Conference on Computer and Games*, 2006.

[4] Freeling, Christian. URL http://www.mindsports.net/CompleteGames/Checkmate/Congo.html.

[5] JiHai, Lee: *WXF or Asia rule*, 2005. URL http://www.wxf.org/xq/rule/asia/asiarule.htm.

[6] Moreland, Bruce: *Computer Chess – Programming Topics*, 2001. URL http://www.seanet.com/∼brucemo/topics/topics.htm.

[7] Osborne, Martin J. und Ariel Rubinstein: *A course in game theory.* MIT Press, 2003.

[8] Ottmann, Thomas und Peter Widmayer: *Algorithmen und Datenstrukturen.* Spektrum, 2002.

[9] Russell, Stuart J. und Peter Norvig: *Artificial Intelligence. A Modern Approach.* Prentice-Hall, 2003.

[10] Schaeffer, Jonathan und Aske Plaat: *New Advances in Alpha-Beta Searching.* In: *ACM Conference on Computer Science*, Seiten 124–130, 1996.

[11] Sutton, Richard S. und Andrew G. Barto: *Reinforcement Learning: An Introduction.* MIT Press, 1998.