UCT: Selektive Monte-Carlo-Simulation in Spielbäumen

Michael Wächter michael_waechter@gmx.de

Abstract

Im Gegensatz zu klassischen Ansätzen wie dem Alphabeta-Algorithmus durchsuchen Monte-Carlo-Methoden in Spielen zufällig Teile des Spielbaums, um damit auf den Wert der Wurzel zu schließen und eine Zugentscheidung zu treffen.
UCT stellt einen neuen, durch (Coulom, 2006) und (Kocsis & Szepesvári, 2007) vorgestellten Monte-Carlo-Algorithmus dar: Zu Beginn durchsuchen die Monte-Carlo-Simulationen den Spielbaum zufällig. Später werden sie dann anhand der bisher berechneten Ergebnisse selektiv durch den Spielbaum geleitet, um schneller zu den gewünschten Ergebnissen zu gelangen.
Neben der Beschreibung des Algorithmus wird auf seine Eigenschaften, Relevanz für Poker, Verbesserungsmöglichkeiten und Resultate in existierenden Systemen eingegangen.

1. Einleitung

Die grundlegendste Möglichkeit, um ein Programm zu befähigen ein bestimmtes Spiel zu spielen, stellen auf Expertenwissen basierende, "handgeschnitzte" Zugentscheidungsfunktionen dar. Beispielsweise benutzte das Pokerprogramm Loki der University of Alberta zu Anfang eine solche Funktion (Billings, Papp, Lourdes, Schaeffer, & Szafron, 1999).
Dieser Ansatz leidet unter vielen Nachteilen. Zum einen ist die Spielstärke des Programms beschränkt durch das Wissen des Experten. Zum anderen ist die Integration von Wissen sehr arbeitsintensiv und fehleranfällig, da Teile des Wissens dazu neigen mit anderen zu interagieren und Programme nur schlecht fähig sind, die Wichtigkeit einzelner "Wissensteile" abzuwägen. In Go beispielsweise ist das benötigte Wissen auch oft zu abstrakt, als dass es direkt in ein Programm umgesetzt werden könnte.
Für einige deterministische Spiele mit vollständiger Information wie Schach und Dame hatte sich bereits relativ früh herausgestellt, dass dieser wissensintensive Ansatz schwierig umzusetzen und wenig von Erfolg gekrönt war. Die überwältigende Mehrheit der Programme und Forschergruppen konzentrierte sich daher auf den suchintensiven Ansatz: Die natürliche Erweiterung statischer Evaluationsfunktionen bestand im Minimax- oder Alphabeta-Algorithmus: Der Spielbaum[1] wird soweit expandiert, wie es die zur Verfügung stehenden Ressourcen zulassen. Nun werden die Blätter mit solch einer auf Expertenwissen basierenden Heuristik evaluiert und ihre Werte zur Berechnung des Werts der Wurzel verwendet. Es stellte sich heraus, dass bereits Evaluationsfunktionen, die den Wert einer Stellung nur grob annähern[2], zu hohen Spielstärken führen können, die natürlich die reine Spielstärke dieses geringen Expertenwissens bei weitem übersteigen. Die Erfolge von Chinook[3] und DeepBlue[4] bewiesen dann eindrucksvoll, dass pure Rechenkraft ausreichte, um

1. Der aktuelle Zustand bildet die Wurzel, mögliche zukünftige Zustände die Knoten und Züge die Kanten.
2. In erster Näherung reicht im Schach z.B. eine Bewertung des noch im Spiel befindlichen Materials.
3. Chinook siegte 1994 über den Weltmeister Tinsley und 2007 wurde das Spiel schließlich komplett gelöst.
4. DeepBlue besiegte 1997 den damals amtierenden Weltmeister Kasparov mit $3\frac{1}{2} - 1\frac{1}{2}$

weltmeisterliches Niveau zu spielen.
Seit den ersten Erfolgen hat sich dieser Ansatz zwar bedeutend weiterentwickelt (transposition tables, iterative deepening, null-move heuristic, killer heuristic, history heuristic, quiescence search). Aber dennoch reicht er nicht aus, um eine Vielzahl an Spielen und damit einhergehenden Problemen zu meistern. Dies kann unterschiedlichste Ursachen haben.
Für indeterministische Spiele (z.B. Poker) beispielsweise kann der Alphabeta-Algorithmus zwar erweitert werden. Allerdings explodiert dann die Größe des Spielbaums und der Algorithmus ist nicht fähig ihn in akzeptabler Zeit zu durchsuchen und brauchbare Ergebnisse zu liefern. Auch KIs für kommerzielle Spiele wie Echtzeitstrategiespiele können auf Grund von hohen Verzweigungsfaktoren und Indeterminismus nicht auf Alphabeta setzen.
Go bringt schon von sich aus einen sehr großen Verzweigungsfaktor im Spielbaum mit. Zudem ist eine Suche mit einer Tiefe von bspw. 12 Zügen schlicht nicht ausreichend, da es zahlreiche Situationen gibt, in denen Menschen fähig sind, die Konsequenzen von Zügen tiefer abzuschätzen. Der möglicherweise wichtigste Grund jedoch ist, dass Züge stark dazu neigen miteinander zu interagieren. Das Spiel ist fast immer dynamisch, selten ruhig und es gibt eine Vielzahl an Faktoren, die über den Wert einer Stellung entscheiden. Es ist daher sehr schwierig, eine Evaluationsfunktion zu schreiben, die den Wert einer Stellung an den Blättern des Alphabeta-Algorithmus adäquat bewertet (vgl. auch (Chaslot, Winands, Uiterwijk, van den Herik, & Bouzy, 2007)).
In (Brügmann, 1993) wurde dann jedoch die Möglichkeit der Anwendung von Monte-Carlo-Methoden für Go in Erwägung gezogen und in (Bouzy & Helmstetter, 2003) wieder aufgegriffen. Die Entwicklung von UCT durch (Coulom, 2006) bzw. (Kocsis & Szepesvári, 2007) führte schließlich dazu, dass die Verwendung von Monte-Carlo-Methoden für Go allgemein Beachtung fand und UCT mittlerweile von vielen der weltweit besten GO-KIs verwendet wird. Außerdem gibt es viele Forschergruppen, die an weiteren Themen zu UCT wie Parallelisierbarkeit (Cazenave & Jouandeau, 2007) oder Integration von Patterns (Gelly & Silver, 2007) arbeiten.
Auch für Poker wurde die Einsetzbarkeit von Monte-Carlo-Methoden erkannt (Billings et al., 1999), die Überlegenheit gegenüber dem zuvor verwendeten Expertenwissenansatz experimentell bestätigt und zahlreiche weitere Vorzüge des neuen Ansatzes hervorgehoben.

Kapitel 2 beschreibt die Funktionsweise von Monte-Carlo-Methoden und speziell UCT.
Kapitel 3 geht auf Eigenschaften des Algorithmus ein.
Kapitel 4 stellt dar, wie UCT für Poker eingesetzt werden kann und welche Vorzüge es gegenüber anderen Ansätzen hat.
Kapitel 5 schließlich versucht aufzuzeigen, welche Möglichkeiten zur Erweiterung von UCT bestehen oder gegenwärtig untersucht werden.
Für ein schnelles Erfassen des Inhalts dieser Ausarbeitung eignet sich am besten das Lesen der Abschnitte 2.2, 4, sowie evtl. 5.1 und 5.2.

2. Funktionsweise

2.1 Monte-Carlo-Algorithmen

Der grundlegende Monte-Carlo-Algorithmus funktioniert folgendermaßen: Zunächst wird der Spielbaum mit konstanter Tiefe 1 expandiert (also alle Folgestellungen der aktuellen

Stellung generiert). Anschließend werden alle Kindknoten der Wurzel (Folgestellungen der aktuellen Stellung) durch Monte-Carlo-Simulationen evaluiert und schließlich der Zug ausgewählt, der zur bestbewerteten Stellung führt.
Den eigentlich wichtigen Teil stellt die MC-Evaluation dar: Von der zu bewertenden Stellung ausgehend werden mehrere (i.A. mehrere 1000) Simulationen durchgeführt. Pro Simulation werden von der zu bewertenden Stellung aus so lange zufällige Züge gespielt, bis man an einer Terminalposition (eine Stellung, in der das Spiel endet) ankommt. Diese kann nun trivial bewertet werden[5]. Im Poker bspw. kann man als Bewertung einfach die gewonnene oder verlorene Geldmenge des Spielers wählen. Hat man mehrere solcher zufälligen Partien durchgeführt, ist der Wert der Stellung der Durchschnittswert aller Simulationen.

Im Vergleich zu allen zuvor in Poker oder Go verwendeten Ansätzen ist an diesem bemerkenswert, dass er mit einem Minimum an Wissen über das Spiel auskommt. Es wird abgesehen von grundlegenden Funktionen nur eine Funktion, die Terminalpositionen bewertet, benötigt.
Dieser Ansatz hatte dennoch unübersehbare Schwächen. Zum einen konnte er Go nicht auf dem selben Niveau wie klassische Ansätze spielen. Zum anderen war er zwar strategisch stark, da er seine "Aufmerksamkeit" breit fächert und stets alle Züge "in Erwägung zog", war aber taktisch schwach, da er nicht fähig war, seine "Aufmerksamkeit" auf interessante, wichtige Züge "zu konzentrieren". Es gab daher verschiedene Versuche dies zu verbessern:
In (Billings et al., 1999) wird die zufällige Zugauswahl in den Simulationen durch Expertenwissen zu den interessanten Zügen gelenkt. Die Auswahl der Züge geschieht also nicht mehr gleichverteilt sondern mit einer durch eine Expertenfunktion vorgegebenen Wahrscheinlichkeit. In Poker schien dieser Ansatz gut zu funktionieren.
In (Bouzy & Helmstetter, 2003) wurde eine weitere Expansion des Spielbaums an der Wurzel versucht. Dies erhöhte jedoch den Rechenaufwand stark und brachte keine wesentliche Spielstärkenverbesserung. In (Bouzy, 2004) wurde Iterative Deepening vorgeschlagen: In jedem Durchlauf wird der Spielbaum eine Ebene tiefer expandiert, die Blätter MC-simuliert und die daraus gewonnnen Informationen für die Beschneidung des Baums im nächsten Durchlauf mit tieferer Suchtiefe verwendet[6]. Laut (Coulom, 2006) schnitt dieser Algorithmus jedoch oft auch brauchbare Äste ab.
In (Coulom, 2006) und (Kocsis & Szepesvári, 2007) wurde schließlich UCT vorgeschlagen[7].

2.2 Upper Confidence Bounds applied to Trees (UCT)

UCT stammt vom UCB-Algorithmus ab, da UCB das Exploration-Exploitation-Dilemma löst und dies auch in Spielbäumen zu lösen ist (siehe hierzu Anhang B).

5. In einem Spiel sollte es am Ende einer Partie trivial sein, den Gewinner und wenn gewünscht die Höhe des Siegs zu ermitteln. Auf Go trifft diese Aussage allerdings nur bedingt zu. Am Ende einer Partie ist es nicht ohne weiteres möglich, den Gewinner zu ermitteln. Es ist zwar möglich die Partie weiter zu spielen bis das Ergebnis trivial ermittelbar ist, allerdings kann das Weiterspielen zu einer anderen Siegöhe und eventuell zu einem anderen Sieger führen.
6. Im 2. Durchlauf mit einem Baum der Tiefe 2 werden also die Äste, die sich im 1. Durchlauf als zu schlecht erwiesen haben, abgeschnitten.
7. (Coulom, 2006) kommt eher von der praktischen Seite der Anwendbarkeit in Spielen. (Kocsis & Szepesvári, 2007) geht vom theoretischen Multiarmed-Bandit-Problem aus und erprobt den Algorithmus auch an eher theoretischen Problemen.

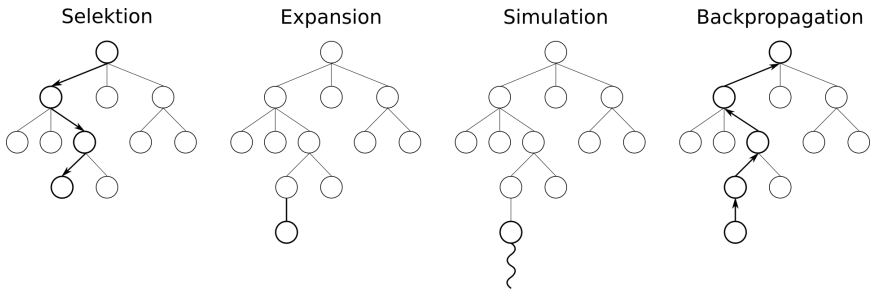

Figure 1: Die vier wesentlichen Teile von UCT. Die Zeichnung ist der Zeichnung in (Chaslot et al., 2007) nachempfunden.

Der Algorithmus expandiert den Spielbaum an der Wurzel nicht mit einer konstanten Tiefe sondern selektiv danach welche Varianten sich als erfolgversprechend herausgestellt haben oder noch herausstellen könnten. Der Baum wird komplett im Speicher gehalten[8] und abgesehen von den unbedingt notwendigen Infos, wie Zeiger auf die Nachfolgerknoten wird in jedem Knoten nur gespeichert, wie viele Simulationen schon durch diesen Knoten gelaufen sind und welchen Wert diese Simulationen summiert hatten. Somit kann später leicht bestimmt werden, welchen Wert jeder Knoten durchschnittlich hat.
Der Algorithmus gliedert sich im Wesentlichen in vier Teile, Selektion, Expansion, Simulation und Backpropagation, die in jeder Simulation nacheinander durchlaufen werden.

2.2.1 Selektion

Von der Wurzel ausgehend wird im Spielbaum ein Weg zu einem Knoten ausgewählt von dem aus der Baum weiter expandiert wird. Bei der Wahl des Knotens muss wie bereits erwähnt abgewogen werden, ob man gute (Exploitation) oder vielversprechende (Exploration) Knoten wählt.
Hierzu wechselt UCT (von der Wurzel ausgehend) an jedem Knoten p zum Kindknoten k mit der höchsten "Upper Confidence Bound": $k = argmax_{i \in I}(v_i + C\sqrt{\frac{\ln n_p}{n_i}})$
I ist die Menge der Kindknoten von p, v_i der Durchschnittswert der Simulationen, die durch i gelaufen sind, und n_p bzw. n_i die Zahl der Simulationen durch p bzw. i.
v_i ist also der bisherige Wert eines Knotens und $C\sqrt{\frac{\ln n_p}{n_i}}$ stellt eine Art Zuversichtlichkeitsbonus dar, der Knoten einen Bonus gibt, je weniger Simulationen bisher durch sie gelaufen sind, also je unsicherer sie sind. Über die Konstante C kann die "Neugierde" des Algorithmus gesteuert werden: Ist C klein, wird der Baum tiefer expandiert, da hauptsächlich die bisher beste Variante untersucht wird. Ist C groß, wird der Baum eher breiter expandiert. Laut (Chaslot et al., 2007) muss C experimentell ermittelt werden, in (Kocsis & Szepesvári, 2007) wird jedoch einfach $\sqrt{2}$ verwendet.

8. Dies führt dazu, dass der Algorithmus leider etwas speicherintensiv ist. Die Größe des Baums ist aber immernoch relativ passabel im Vergleich dazu wenn man bspw. einen kompletten Alphabeta-Baum im Speicher behalten wollte.

2.2.2 Expansion

Gelangt die Selektion an einem Knoten an, für den bisher weniger als ein Schwellwert T an Kindknoten im Baum expandiert wurden (im Spezialfall also ein Blatt), wird hier weiter expandiert, dem Knoten also weitere Kindknoten hinzugefügt.
Die einfachste denkbare Strategie hierbei ist, einen zufälligen neuen Kindknoten hinzuzufügen. Hier sind aber auch intelligentere Ansätze möglich. In (Chaslot et al., 2007) bspw. wird zu Anfang jeweils nur ein zufälliges Kind hinzugefügt. Durchläuft man schließlich den Knoten einmal wenn er bereits T Kinder hat, werden sofort alle noch fehlenden Kinder hinzugefügt. Es sind aber auch Strategien denkbar, die Expertenwissen verwenden und z.B. Schlagzüge zuerst expandieren.

2.2.3 Simulation

In der Simulation wird der Wert eines neu hinzugefügten Knotens abgeschätzt. Genauso wie beim reinen Monte-Carlo-Algorithmus wird hierzu vom Knoten aus eine simulierte Partie bis zu einem Terminalknoten gespielt. Dies geschieht im einfachsten Fall durch rein zufällige Züge, kann aber auch durch eine geeignete Simulationsstrategie (vgl. Abschnitt 5.1) gesteuert werden. Hier bieten sich bspw. Patterns (Gelly, Wang, Munos, & Teytaud, 2006; Gelly & Silver, 2007) oder die bereits bei den Monte-Carlo-Ansätzen erwähnte Strategie in (Billings et al., 1999) an.
Wird eine Simulationsstrategie verwendet, kann sich zwar das Spielniveau deutlich steigern, allerdings steigt auch die Rechenzeit des Algorithmus um evtl. mehrere Größenordnungen (vgl. (Chaslot et al., 2007)). Ein Ausweg ist, dass die Simulationsstrategie eventuell nur für wichtige (z.B. ausreichend oft besuchte) Knoten verwendet wird. Außerdem muss beachtet werden, dass die Strategie weder zu randomisiert noch zu selektiv vorgehen darf, da in beiden Fällen offensichtlich die Spielstärke leidet.
Der Wert des zu simulierenden Knotens entspricht dem Wert der durchgeführten Simulation. Ob dies die Höhe des Siegs oder einfach nur +1 für einen Sieg und -1 für eine Niederlage sein soll, muss abgewogen werden[9].

2.2.4 Backpropagation

Nach der Simulation muss das ermittelte Ergebnis im Baum nach oben bis zur Wurzel durchgereicht werden.
In der Standardversion wird einfach der Durchschnittswert aller durch den Knoten durchgelaufenen Simulationen nach oben gereicht. Wie bereits erwähnt enthält jeder Knoten Informationen, wie viele Simulationen durch ihn durchgelaufen sind und welches Ergebnis sie summiert hatten. Der Wert v_i eines Knotens in der Selektionsformel (s.o.) ist dann offensichtlich die Summe der Werte geteilt durch die Zahl der Simulationen.
(Coulom, 2006) zeigt experimentell, dass dieser "Mean"-Backupoperator dazu neigt, den wahren Wert eines Knotens zu unterschätzen. Eine Alternative, der "Max"-Backupoperator, der den Wert des Kindes mit dem höchsten Wert hochreicht, überschätzt hingegegen den

9. In Go bspw. ist die Höhe des Siegs unerheblich. Die Höhe des Siegs einfließen zu lassen, kann die KI dazu veranlassen gierig zu spielen und das Spiel damit zerbrechlich, also anfälliger gegenüber gegnerischen Atacken werden zu lassen. Liegt man vorne, reicht es aus sicher zu spielen und nicht noch die Höhe des Siegs zu vergrößern. Im Poker ist dieser Sachverhalt natürlich noch einmal ganz anders.

wahren Wert. Der "Robust Max"-Operator reicht den Wert des Kindes, durch das die meisten Simulationen gelaufen sind, hoch. Für kleine Simulationszahlen liefert er zwar schlechtere, für große Simulationszahlen hingegen deutlich bessere Vorhersagen. Der beste Operator ist jedoch "Mix": Eine Linearkombination aus "Mean" und "Robust Max" (sowie ein paar Verfeinerungen für Spezialsituationen). Er konvergiert für hohe Simulationszahlen zwar nur wenig besser als "Robust Max", liefert allerdings für niedrige Simulationszahlen wesentlich bessere Vorhersagen.
Da für "Mix" jedoch die Koeffizienten der Linearkombination optimiert werden müssen, ist für eine einfache Implementierung evtl. doch "Mean" oder "Robust Max" vorzuziehen.

2.2.5 Zugauswahl

Hat man die gewünschte Anzahl an Iterationen durchgeführt, befindet sich ein selektiv expandierter Baum im Speicher und man muss aus ihm einen Zug auswählen. Die meistverwendete Möglichkeit ist es, an der Wurzel den Zug zu wählen, durch den die meisten Simulationen gelaufen sind.

3. Eigenschaften und Ergebnisse

3.1 Konvergenz

In (Kocsis & Szepesvári, 2007) wird unter anderem gezeigt, dass der Algorithmus in seiner grundlegendsten Form (bspw. ohne Simulationsstrategie) die Fehlerwahrscheinlichkeit an der Wurzel für $t \to \infty$ polynomiell gegen 0 gehen lässt. Der Algorithmus konvergiert also gegen das korrekte Resultat, das auch eine vollständige Alphabeta-Berechnung bis hin zu den Terminalknoten geliefert hätte.

3.2 Expertenwissen

Ein bereits hervorgehobener Vorzug von UCT ist, dass in der grundlegendsten Form kein Expertenwissen ausser einer (meist trivialen) Funktion, die Terminalknoten bewertet, benötigt wird. Insbesondere wird, da immer bis zu Terminalknoten simuliert wird, nicht wie in Alphabeta eine Evaluationsheuristik für Nichtterminalknoten benötigt. In der Go-Programmierung hatte sich nämlich genau dies als Problem herausgestellt (vgl. Abschnitt 1).
In UCT dient sämtliches Wissen nur noch dem Zweck der Spielstärkensteigerung. In (Billings et al., 1999) wird auch erwähnt, dass sich bei Monte-Carlo-Ansätzen verwendetes Expertenwissen sehr gut kapseln lässt (was auf UCT auch zutreffen dürfte), was die Entwicklungszeit und Wartbarkeit verbessert.
Bei der Integration von Expertenwissen muss aber wie bereits erwähnt die Spielstärkenverbesserung mit der Rechenzeiterhöhung abgewogen werden. Ist das Wissen gut gekapselt, sollte es aber auch möglich sein an seiner Performanz zu arbeiten ohne die Lesbarkeit des Gesamtalgorithmus merklich zu verschlechtern.

3.3 "Anytime"-Eigenschaft

Der Algorithmus ist "jederzeit". Das heißt konkret, dass die Iterationen jederzeit abgebrochen werden können und das Ergebnis (der Baum samt Bewertungen) trotzdem brauch-

bar ist. Bei einem reinen Alphabeta-Algorithmus wäre das Ergebnis nach einem Abbruch höchstwahrscheinlich unbrauchbar. Zu diesem Zweck wurde für Alphabeta zwar die Technik des Iterative Deepening erfunden[10]. Nichtsdestotrotz sind die Ergebnisse der letzten Iteration unbrauchbar und da sie wegen exponentiellem Rechenzeitanstieg die meiste Rechenzeit in Anspruch genommen hat, ist der Großteil der Rechenzeit "verschwendet"[11]. Bei UCT ist dies alles nicht der Fall.
Die Möglichkeit den Algorithmus quasi jederzeit abbrechen zu können ist nicht nur generell bei Zeitdruck vorteilhaft. Es ist auch denkbar, dass der Algorithmus versucht komplizierte und einfache Stellungen zu erkennen und ihnen entsprechend eine größere oder kleinere Anzahl an Simulationen zugesteht. Außerdem ist es mit Pondering (vgl. Abschnitt 5.7) möglich "mitzudenken" solange der Gegner über seinen Zug nachdenkt und die Berechnungen abzubrechen, sobald er seinen Zug macht.

3.4 Ruhesuche

Im Alphabeta-Algorithmus besteht das sog. Horizontproblem[12]. Behoben wird es mit der sog. Rugesuche: Varianten, die dazu neigen, das Ergebnis der Partie stark zu beeinflussen (z.B. Schlagzüge), werden noch einmal tiefer als normal expandiert, bis die Stellung sich "beruhigt" hat.
In UCT besteht dieses Problem grundsätzlich nicht, da alle Varianten bis zu Terminalpositionen hin ausgespielt werden. Sollte einem auffallen, dass bestimmte wichtige Varianten von den Simulationen zu selten berücksichtigt werden, so sollte auf jeden Fall die Integration einer Simulationsstrategie (s. Abschnitt 5.1) in Erwägung gezogen werden. Diese sollte dann die Simulation der bisher zu häufig "durchs Raster gerutschten" Varianten besonders hoch gewichten.

3.5 Strategie vs. Taktik

Wie bereits in Abschnitt 2.1 erwähnt, neigen Monte-Carlo-Methoden dazu strategisch stark und taktisch schwach zu spielen. UCT gleicht dieses Problem bis zu einem gewissen Grad aus, da es selektiv die "interessanten" (die guten und die vielversprechenden) Varianten untersucht. Dennoch ist es gut möglich, dass der Algorithmus in Reinform große taktische Schwächen aufweist.
Auch dies kann durch Simulationsstrategien eliminiert werden. Außerdem kann auch der Selektionsteil (und speziell die Selektionsformel) erweitert werden. Beispiele sind Progressive Bias (Abschnitt 5.5) und Patterns (Abschnitt 5.1). Manche Ansätze (z.B. Progressive Unpruning (Abschnitt 5.4)) kommen dabei sogar ohne Expertenwissen aus.

10. Der Baum wird zunächst mit Suchtiefe 1 durchsucht, anschließend mit Tiefe 2 usw.. Die Ergebnisse einer Iteration können benutzt werden, um in der nächsten Iteration eindeutig schlechte Äste abzuschneiden und die Suche zu beschleunigen. Wird der Algorithmus abgebrochen gilt das Ergebnis der letzten vollständigen Iteration.
11. Natürlich sind die Berechnungen nicht ganz vergeblich. Bspw. wurden durch die letzte Iteration die transposition tables mit wertvollen Informationen gefüllt.
12. Der Algorithmus neigt dazu unvermeidbare aber nach hinten verschiebbare schlechte Züge (z.B. einen drohenden Damenverlust) über seinen Suchhorizont hinauszuschieben, sodass er sie nicht mehr sieht.

3.6 Ergebnisse in realen Systemen

In (Billings et al., 1999) wurde der Austausch eines wissensbasierten durch einen suchintensiven Ansatz (MC mit Simulationsstrategie aber ohne UCT) im Pokerprogramm Loki beschrieben. Das Ergebnis war eine signifikante Verbesserung (mind. 0.05 sb/hand).
UCT selbst wurde bisher auch in diversen Anwendungsfällen erprobt: Seine Entdeckung führte zu einer starken Beachtung von MC-Methoden im Computer-Go. UCT-basierte Programme, sind fähig mit wissensbasierten Programmen, in denen mehrere Mannjahre Entwicklung stecken, mitzuhalten. In 9x9-Go sind fünf der sechs besten Programme UCT-basiert, in 19x19-Go dominieren im Moment noch wissensbasiertn Programme, den UCT-Programmen wird aber ein großes Verbesserungspotenzial prognostiziert (Bouzy, 2007).
In (Kocsis & Szepesvári, 2007) wird die Leistungsfähigkeit UCTs an eher künstlichen Problemen erprobt: Das erste Experiment waren zufällige Spielbäume: Spielbäume von Zweipersonenspielen (deterministisch und mit vollständiger Information) werden zufällig erzeugt und ihr Wert muss von UCT ermittelt werden. UCT hat sich hierbei als Alphabeta überlegen herausgestellt, obwohl Alphabeta bisher der Standardansatz für deterministische Spiele mit vollständiger Information war. Auch legen die Ergebnisse nahe, dass UCT schneller als $o(B^{D/2})$ (B ist der Verzweigungsfaktor und D die Tiefe des Baums) konvergiert. Es lassen sich also (wie bei Alphabeta vs. Minimax) doppelt so tiefe Spielbäume evaluieren wie mit reinem MC-Ansatz. Das zweite Experiment war das Stochastic Shortest Path Problem: Ein Segelboot muss unter wechselnden Windbedingungen ein Ziel erreichen. Hier war UCT insbesondere bei großen Problemen (kleine Planungsraster-Größe) wesentlich erfolgreicher als die bisher auf dieses Problem speziell zugeschnittenen Algorithmen. Bei den großen Problemen waren die herkömmlichen Algorithmen zumeist garnicht fähig in akzeptabler Zeit ein Ergebnis zu liefern.

4. Relevanz für Poker

4.1 Nötige Adaptionen

Für Pokern müssen ein paar Adaptionen gemacht werden [13]: Es muss unvollständiges Wissen über den aktuellen Spielstand (verdeckte Karten etc.), Nichtdeterminismus (zufälliges Austeilen von Karten) und die Beteiligung mehrerer Spieler berücksichtigt werden.

4.1.1 Nichtdeterminismus

Die Integration von Nichtdeterminismus ist vergleichsweise einfach. Hierzu werden in den Spielbaum Ebenen zwischen den Ebenen, zwischen denen nichtdeterministische Aktionen stattfinden, eingezogen[14]. Es muss allerdings beachtet werden, dass in der Selektion und der Simulation für die Kanten, die nichtdeterministische Aktionen repräsentieren, keine

13. Diese sind z.T. in (Kocsis & Szepesvári, 2007) in der abstrakten Darstellung als Markovian Decision Problem schon berücksichtigt
14. Wenn z.B. Karten gegeben werden und anschließend eine Entscheidung eines Spielers ansteht, wird eine weitere Ebene eingezogen. Die Kanten zu dieser Ebene hin stellen das Geben der Karten und die Kanten von dieser Ebene weg die Entscheidungen des Spielers dar.

Heuristiken (vgl. Abschnitt 5.1) verwendet werden. Die Wahl der "nichtdeterministischen Kanten" muss mit der selben Wahrscheinlichkeit geschehen wie im realen Spiel[15].

4.1.2 Unvollständiges Wissen

Die einfachste Möglichkeit dies zu berücksichtigen, ist es das fehlende Wissen wie bei einer nichtdeterministischen Aktion "auszuwürfeln". Danach kann es als bekannt vorausgesetzt werden. Zum Beispiel kann direkt am Anfang des Spielbaums wie oben beschrieben eine "nichtdeterministische Ebene" eingezogen und die Karten der Gegenspieler zufällig festgelegt werden.
Kompliziertere Ansätze führen z.B. Protokoll über den bisherigen Spielverlauf (sowohl das Spielen vorheriger Hände als auch das Spielen der aktuellen Hand) und ermitteln damit ein Wahrscheinlichkeitsprofil der Karten jedes Spielers. In (Billings et al., 1999) wird dieser Ansatz verfolgt.

4.1.3 Mehrspielerspiel

Klassische Algorithmen berücksichtigen nur Zweipersonennullsummenspiele[16]. Eine Möglichkeit zur Erweiterung für mehrere Spieler ist folgende: Zunächst werden natürlich Ebenen für die Entscheidungen jedes Spielers im Spielbaum vorgesehen (zzgl. der "nichtdeterministischen Ebenen" (s.o.)). Die Funktion, die die Terminalpositionen bewertet, muss nun statt positiven/negativen Werten für gewonnene/verlorene Spiele Tupel, die die Gewinne und Verluste jedes Spielers enthalten, zurückgeben. Im Backpropagating wird dann das gesamte Tupel nach oben weitergereicht und auf das bisherige Tupel aufaddiert. In der Selektion muss nun nicht mehr der Durchschnittswert der Partie (und der Zuversichtlichkeitsbonus) maximiert werden, sondern ein anderer Ausdruck. Hier bietet sich die Differenz zwischen dem eigenen Gewinn und dem summierten oder durchschnittlichen Gewinn der Gegenspieler an. Im Spezialfall mit zwei Spielern entspricht dies wieder der ursprünglichen Maximierung.

4.2 Implizites Wissen

Einer der Hauptvorzüge von Monte-Carlo-Ansätzen allgemein (und natürlich speziell UCT) ist wie bereits erwähnt, dass grundsätzlich kein Wissen über das Spiel nötig ist.
In (Billings et al., 1999) wird bereits hervorgehoben, dass die Modellierung von Konzepten wie Handstärke, Handpotenzial und "Pot odds", sowie auch subtilere Konzepte wie "implied odds", nicht explizit geschehen muss. Sie sind bereits implizit in der Simulation enthalten. Starke Hände werden sich in der Mehrzahl der Simulationen auch als stark herausstellen und hoch bewertet werden. Für Hände mit Potenzial trifft das selbe zu.
Der Vorteil dieses impliziten Wissens ist wie bereits erwähnt leichte Kapselbarkeit des Expertenwissens, kurze Entwicklungszeiten, leichte Erweiterbarkeit, geringer Wartungsaufwand, keine Spielstärkenbeschränkung durch die Stärke des integrierten Expertenwissens uvm..

15. Es sei denn man möchte auf diese Weise Spielerprofile integrieren (vgl. Abschnitt 5.2)
16. Der Gewinn des einen Spielers ist der Verlust des anderen.

5. weitere Anwendungs- und Erweiterungsmöglichkeiten

5.1 Simulationsstrategien und Integration von Offline-Wissen

Simulationsstrategien sind scheinbar die Hauptforschungsrichtung in der momentan versucht wird UCT zu erweitern. Sie lassen die MC-Simulation an den Blättern des Baums nicht rein zufällig ablaufen, sondern steuern sie basierend auf handcodiertem Expertenwissen oder zuvor automatisch generiertem Wissen (Offlinewissen). Diese Technik ist natürlich vollständig von der jeweiligen Domäne (Poker, Go, ...) abhängig.
In Go können dies bspw. (automatisch aus Spieldatenbanken oder per Hand von Experten erzeugte) Zugpatterns, die Anzahl an gefangenen Steinen, die Nähe zum vorherigen Zug und vieles mehr sein (siehe (Chaslot et al., 2007; Gelly & Silver, 2007; Gelly et al., 2006) u.v.a.). Auch andere statische Ansätze, die in der Zeit vor UCT entstanden sind, wie gezüchtete KNNs zur automatischen Zuggenerierung (Donnelly, Corr, & Crookes, ; Richards, Moriarty, McQuesten, & Miikkulainen, 1997), sind denkbar und leicht integrierbar.
Strategien zur Simulation wurden für Poker bereits in (Billings et al., 1999) vorgestellt. Dort wird allerdings noch eine reine MC-Simulation ohne UCT durchgeführt. Die Ergebnisse sind trotz dessen allerdings schon beachtlich. Strategien können nicht nur in der Simulation sondern wie bei Progressive Bias (s.u.) auch in der Selektion benutzt werden.
Da die Verwendung von heuristischem Wissen zu einer Stellung meist sehr rechenintensiv ist, müssen Rechenzeit und Leistungsverbesserung stets vorsichtig abgewogen werden. Auch kann man (wie dies in (Chaslot et al., 2007) geschieht) die Berechnung gewisser Teile der Heuristik an- oder abschalten, je nachdem wie wichtig eine korrekte heuristische Evaluation des aktuellen Knotens ist.

5.2 Integration von Spielerprofilen

Die Integration von Spielerprofilen kann auf vielfache Weise geschehen. Beispielsweise kann in der normalen UCT-Selektionsformel (Abschnitt 2.2.1) ein weiterer Summand hinzugefügt werden, der wissensbasiert je nach Spieler unterschiedliche Züge bevorzugt. Auch ist es denkbar je nach Spieler die Werte von Knoten zu modifizieren, um Optimismus/Pessimismus oder Gier/Vorsicht zu modellieren. In der Simulationsstrategie kann ein Spielerprofil integriert werden, indem ebenfalls je nach Spieler bestimmte Züge bevorzugt werden (sowohl für "deterministische" als auch für "nichtdeterministische Kanten", vlg. Abschnitt 4.1.1).

5.3 Parallelisierbarkeit

In (Cazenave & Jouandeau, 2007) wird die Parallelisierbarkeit von UCT behandelt. Die einfachste Möglichkeit ist es, mehrere Prozessoren unabhängig an der selben Stellung rechnen zu lassen und am Ende die jeweiligen Ergebnisse an der Wurzel und ihren Kindknoten zu kombinieren und davon ausgehend den Zug zu wählen. Dieser Ansatz benötigt sehr wenig Kommunikation der Algorithmen untereinander und umgeht das Problem eines gemeinsamen Spielbaums, da jeder Prozessor einen eigenen pflegt.
Neben dieser sog. "Single-Run"-Parallelisierung werden auch noch "Multiple-Runs" und "At-the-leaves" vorgeschlagen. Sie resultieren in einer höheren Spielstärke, erfordern aber mehr Kommunikation der Prozessoren.

5.4 Progressive Widening / Unpruning

In (Chaslot et al., 2007) wird die Technik des Progressive Unpruning vorgeschlagen: Wurde ein Knoten weniger als T mal besucht ($n_p < T$), werden nacheinander alle seine Kinder erweitert und simuliert. Ist $n_p = T$, werden zunächst die meisten der Kinder temporär abgeschnitten. Es wird also angenommen, dass Zeitmangel herrscht und nur wenige Knoten werden weiter untersucht. Steht irgendwann dann doch wieder mehr Zeit zur Verfügung ($n_p > T$) werden wieder sukkzessive Knoten hinzugefügt (Unpruning) und simuliert. Ist irgendwann $n_p >> T$ sind wieder alle Knoten vorhanden.

5.5 Progressive Bias

Ebenfalls in (Chaslot et al., 2007) vorgeschlagen wird Progressive Bias: Wie in der Simulationsstrategie soll auch in der Selektion die Wahl von heuristisch als gut bewerteten Stellungen bevorzugt werden. Der Einfluss dieser Heuristik soll jedoch mit steigender Zahl an Simulationen, die durch diesen Knoten gelaufen sind, an Bedeutung verlieren, da dann eher das Ergebnis der Simulationen akkurater als die Heuristik ist. Hierzu wird der normalen UCT-Selektionsformel ein Summand $f(n_i) = \frac{H_i}{n_i+1}$ hinzugefügt. H_I stellt das heuristische Wissen dar. Mit steigender Simulationszahl n_i wird $f(n_i)$ bedeutungsloser.
Insbesondere in Kombination mit Progressive Unpruning hat diese Technik beachtliche Ergebnisse geliefert. Gegen ein Vergleichsprogramm (GnuGo 3.6) stieg die Siegrate im 19x19-Go von 0% auf 48.2%.

5.6 Transposition Tables

(Bouzy, 2007) schlägt eine Technik vor, die schon in der Schach-Programmierung benutzt wurde: Transposition Tables. Sowohl in Poker als auch in Go gibt es häufig äquivalente Knoten im Spielbaum. Bspw. kann ein und die selbe Stellung auch durch die Vertauschung der Zugabfolge erreicht werden. Man kann daher Hashtabellen benutzen, in denen die Ergebnisse vorheriger Knotenevaluationen gespeichert werden. Diese Ergebnisse werden dann an anderen Stellen im Baum wiederbenutzt. Hashtabellen sollte man allerdings vorsichtig benutzen. Bspw. sollte man nur wichtige (häufig besuchte) Knoten speichern, damit die Hashtabelle sich nicht zu sehr mit unrelevanten Daten füllt und auch ein Nachschlagen in der Tabelle sollte nicht zu häufig geschehen (z.B. nur nahe der Wurzel).

5.7 Pondering

Eine weitere Technik aus der Schachprogrammierung ist das Pondering: Damit die Zeit, in der der Gegenspieler denkt, nicht ungenutzt verstreicht, wird auch in dieser Zeit gerechnet. Auf Grund der “Anytime”-Eigenschaft (vgl. Abschnitt 3.3) kann, sobald der Gegner zieht, problemlos die Berechnung abgebrochen und mit der eigentlichen Berechnung begonnen werden.
Im einfachsten Fall benutzt man dann in der eigentlichen Berechnung nur die während des Pondering angefüllten Transposition Tables. In UCT wäre es jedoch auch denkbar, dass man den beim Pondering aufgebauten Baum wiederverwendet und die Wurzel und die durch den Zug des Gegenspielers nicht gewählten Kindknoten der Wurzel abschneidet.

5.8 Killer/History Heuristics

Ebenfalls aus Schach bekannt sind Killer/History Heuristics: Züge, die in einem Teil des Baums gut waren, sind möglicherweise auch in anderen Teilen des Baums erfolgreich. Daher kann eine Liste von erfolgreichen Zügen mitgeführt werden und diese in der Selektion oder Simulation bevorzugt werden.

5.9 "Mercy" rule

(Bouzy, 2007) schlägt die sog. "Mercy" rule vor: Ist in einem Simulationslauf das Ergebnis für einen der beiden Spieler bereits hinreichend schlecht, wird diese Variante als für den anderen Spieler gewonnen angenommen und die Simulation an diesem Punkt nicht mehr weitergespielt.
Diese Heuristik lässt sich allerdings nicht ohne weiteres auf Poker anwenden.

5.10 Weitere Anwendungsmöglichkeiten

In (Chung, Buro, & Schaeffer, 2005) wird eine sehr interessante Anwendung von MC-Methoden vorgeschlagen: Planung in Echtzeitstrategiespielen. Das Konzept von abwechselnden Zügen der Spieler wird hierzu aufgeweicht. Die Spieler können quasi gleichzeitig Züge durchführen und die Autoren schlagen vor, MC-Simulation in allen Hierarchien der Spielplanung einzusetzen (Micro Management, Schlachttaktik, Gesamtpartiestrategie). Für hohe Abstraktionsebenen (z.B. zur Planung der Gesamtstrategie) werden hierzu Elementarzüge (z.B. Vorwärtsbewegen einer Einheit um ein Feld) zu größeren Zügen zusammengefasst. Dies kann entweder vorab durch einen Programmierer oder während des Spiels automatisch (z.B. Zusammenfassung mehrerer Elementarbewegungen durch einen Wegfindungsalgorithmus) oder auch randomisiert geschehen.
Die Ergebnisse waren teilweise noch handgecodeten Programmen unterlegen, der Ansatz selbst ist aber vielversprechend und er wurde noch nicht in Kombination mit UCT erprobt. Die Brauchbarkeit von UCT scheint mit Spielen wie Go und Poker also noch lange nicht erschöpft zu sein und vermutlich wurden viele Anwendungsfelder bisher noch nicht einmal in Betracht gezogen.

Appendix A. Alpha-Beta-Algorithmus

Alphabeta stellt den Standardalgorithmus für Spiele wie z.B. Schach dar. Er wird hier nur kurz zur Vollständigkeit erwähnt. Der folgende Pseudocode entstammt dem Wikipediaartikel (Wikipedia, 2008).

```
int NegaMax(int tiefe, int alpha, int beta){
    if (tiefe == 0)
        return Bewerten();
    GeneriereMoeglicheZuege();
    while (ZuegeUebrig()){
        FuehreNaechstenZugAus();
        wert = -NegaMax(tiefe-1, -beta, -alpha);
        MacheZugRueckgaengig();
```

```
        if (wert >= beta)
            return beta;
        if (wert > alpha)
            alpha = wert;
    }
    return alpha;
}
```

Der Spielbaum wird bis zu einer festen Tiefe expandiert und die Blätter mit einer statischen Bewertungsfunktion bewertet. Anschließend werden die Bewertungen im Baum nach oben durchgereicht, indem in jeder Ebene der Wert des jeweils besten Zugs des in dieser Ebene am Zug befindlichen Spielers nach oben zurückgegeben wird (**alpha** stellt hier den Wert des besten Zugs dar). Dies an sich ist noch äquivalent zum Minimax-Algorithmus. Der Trick besteht in der Zeile **if (wert >= beta) return beta;**. Hier werden Äste des Baums abgeschnitten, die garnicht durch perfektes Spiel auf beiden Seiten zustande gekommen sein können und damit nicht betrachtet werden müssen.

Appendix B. Multiarmed-Bandit-Problem, Exploration vs. Exploitation und UCB

Das Multiarmed-Bandit-Problem sieht folgendermaßen aus: Der Spieler steht vor mehreren Kasinoautomaten, die alle eine unterschiedliche Gewinnwahrscheinlichkeit haben, die man nicht kennt und das Ziel ist es in mehreren Spielen einen möglichst hohen Gewinn zu erzielen. Das Dilemma besteht nun darin entweder am Automaten zu spielen, der bisher die höchste Gewinnwahrscheinlichkeit hatte (Exploitation) oder an anderen Automaten zu spielen, um festzustellen, dass diese eventuell doch eine höhere Gewinnwahrscheinlichkeit haben (Exploration). UCB löst dieses Problem indem es stets den Automaten mit der höchsten upper confidence bound wählt (für Details siehe (Kocsis & Szepesvári, 2007)).
In Spielbäumen besteht nun ein ähnliches Dilemma: Entweder kann der Algorithmus die bisher beste Variante im Baum genauer und tiefer evaluieren (Exploitation) oder er versucht rauszufinden, ob es nicht eventuell doch noch bessere Varianten als die bisher beste gibt (Exploration). In (Kocsis & Szepesvári, 2007) wird daher UCB auf Bäume zu UCT erweitert und bewiesen, dass die Konvergenz des Algorithmus hin zum optimalen Resultat erhalten bleibt.

References

Billings, Papp, Lourdes, Schaeffer, & Szafron (1999). Using Selective-Sampling Simulations in Poker. In *Proceedings of the AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information.*

Bouzy (2004). Associating shallow and selective global tree search with Monte Carlo for 9x9 Go. In *Proceedings Fourth International Conference on Computers and Games.*

Bouzy (2007). Old-fashioned Computer Go vs Monte-Carlo Go.. Online verfügbare Präsentation: `http://ewh.ieee.org/cmte/cis/mtsc/ieeecis/tutorial2007/Bruno_Bouzy_2007.pdf`.

Bouzy, & Helmstetter (2003). Monte Carlo Go developments. In *Proceedings of the 10th Advances in Computer Games Conference*.

Brügmann (1993). Monte Carlo Go..

Cazenave, & Jouandeau (2007). On the Parallelization of UCT. In *Proceedings CGW 2007*.

Chaslot, Winands, Uiterwijk, van den Herik, & Bouzy (2007). Progressive strategies for monte-carlo tree search. In *Proceedings of the 10th Joint Conference on Information Sciences*.

Chung, Buro, & Schaeffer (2005). Monte Carlo planning in RTS games. In *Proceedings of CIG 2005*.

Coulom (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings Computers and Games (CG-06)*. Springer.

Donnelly, Corr, & Crookes. Evolving Go playing strategy in neural networks..

Gelly, & Silver (2007). Combining online and offline knowledge in UCT. In *Proceedings ICML 2007*.

Gelly, Wang, Munos, & Teytaud (2006). Modifications of UCT with Patterns in Monte-Carlo Go..

Kocsis, & Szepesvári (2007). Bandit based Monte-Carlo Planning. In *Proceedings ECML-07*. Springer.

Richards, Moriarty, McQuesten, & Miikkulainen (1997). Evolving neural networks to play Go. In *Proceedings of the 7th International Conference on Genetic Algorithms*.

Wikipedia (2008). Alpha-Beta-Suche.. Der deutsche Wikipediaartikel zum Alphabeta-Algorithmus `http://de.wikipedia.org/wiki/Alpha-Beta-Suche`; Stand 30. März 2008, 14:15.