UCT Simulation in Go

Marian Wieczorek

Betreuer: Prof. Dr. Fürnkranz

25. Juli 2009

Hiermit versichere ich, die vorliegende Bachelor Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

(Ort, Datum)	(Unterschrift)

Inhaltsverzeichnis

Zusammenfassung

Diese Bachelor-Arbeit beschreibt eine Implementierung einer künstlichen Intelligenz für das Spiel *Go*. Die Beschreibung der Implementierung umfasst die Repräsentation des Spielzustands und den verwendeten Algorithmus. Das Programm wählt Züge mit der *UCT*-Suche. Für die Suche werden einige Auswahlheuristiken untersucht. Die Heuristiken werden *First-Play-Urgency* genannt. Ziel der Arbeit ist es, die gewählten Heuristiken gegenüber zu stellen.

Kapitel 1

Einleitung

Go ist ein Spiel, das seit tausenden Jahren gespielt wird. In dieser Zeit sind vielfältige Strategien entstanden. Im Bereich Knowledge Engineering bietet Go aufgrund der wenigen, einfachen Regeln und der freien Zugwahl eine herausfordernde Grundlage für Forschung. Forschungsbereiche beschäftigen sich mit Mustererkennung und künstlichen Intelligenzen.

1.1 Motivation

In dieser Arbeit wird der Algorithmus *UCT* (Upper Confidence bound applied to Trees) untersucht. Mit diesem ist es möglich, Go ohne weitreichendes Expertenwissen zu spielen. UCT ist ein Algorithmus, der das *Monte-Carlo*-Verfahren mit einer *Baumsuche* kombiniert. (Brügmann, 1993) verwendete zum ersten Mal für ein Go-Programm das Monte-Carlo-Verfahren. Schon auf dem Stand der Technik von 1993 erreichte der Spieler mit zufälligen Simulationen die Stärke eines Anfängers. Dieser Schritt legte den Grundstein für viele erfolgreiche Go Programme. CrazyStone von (Coulom, 2006) setzte sich unter der Verwendung von zufallsbasierten Simulationen lange durch. CrazyStone gewann Turniere 2007 und 2008 und besiegte das Programm „Many Faces of Go“ und den Meister Kaori Aoba (4 Dan). Gegen Kaori Aoba bekam CrazyStone sieben Steine Vorsprung. Eine ähnliche Idee verfolgen (Gelly, Wang, Munos, & Teytaud, 2006) mit MoGo. Sie ersetzten die Auswahlheuristik von CrazyStone durch die UCB1-Regel und entwickelten damit UCT. Unter Verwendung einfacher Patterns konnte UCT weiter verbessert werden. Am 20. September 2008 schlug MoGo den Meister Kim Myung Wan (8 Dan). Dabei wurden MoGo sieben Steine Vorsprung gegeben.

Das Monte-Carlo-Verfahren versucht - wie die Kasinos des gleichnamigen Stadtteils von Monaco - Gewinn aus dem Zufall zu ziehen. Die Monte-Carlo-

Suche verwendet zufällige Simulationen, um ein Problem zu lösen. Durch eine Simulation wird ein zufälliges Beispiel des Problems erzeugt. Während das System schwer zu beschreiben ist, muss das Beispiel einfach zu bewerten sein. Nach vielen Iterationen entsteht eine große Menge aus Beispielen mit guten und schlechten Bewertungen. Aus der Verteilung der gewichteten Beispiele wird die Verteilung für den Folgezustand des Gesamtsystems approximiert. Häufig ist es möglich gute Rückschlüsse von elementaren Beispielen auf das Gesamtsystem zu machen. Monte-Carlo-Methoden werden daher vor allem in der Physik und der Chemie zur Simulation von Partikelsystemen verwendet. Aber auch in der Informatik spielt die Simulation eine große Rolle. So findet die Monte-Carlo-Suche zum Beispiel in der Computer Vision, der Robotik (Dellaert, Fox, Burgard, & Thrun, 1999) und im maschinellen Lernen erfolgreiche Anwendungen (Pengelly, 2002). In dieser Arbeit wird der Monte-Carlo-Algorithmus dazu verwendet die Trainingsmenge zu generieren. Im Fall von Go entspricht ein Trainingsbeispiel einem beendeten Spiel. Die Endstellungen können einfach bewertet werden. Durch das Spielen vieler, zufälliger Spiele wird die Verteilung der besten Zugfolgen für beide Gegner geschätzt. Aus dieser Verteilung wählt der Spieler seine Strategie. Die Strategie wird nach jedem Zug neu angepasst.

Es ist möglich allein mit Monte-Carlo eine Strategie zu ermitteln. (Brügmann, 1993) verwendet das Verfahren, um die Zugreihenfolge zu bestimmen. Ein Problem bei diesem Ansatz ist, dass nur eine Liste von Zügen verwaltet wird. Allerdings erkannte (Brügmann, 1993), dass einige Züge nur im Zusammenhang mit anderen gut sein können. Dieser Gedanke führt dazu, die Strategien im Spielbaum zu untersuchen.

Monte-Carlo ist allerdings nicht einfach auf Baumprobleme übertragbar. UCT gibt eine Lösung für dieses Problem. Mit einer Baumsuche wählt UCT einen Knoten und baut den Spielbaum iterativ auf. Die gewählten Knoten werden daraufhin simuliert. Die ursprüngliche Baumsuche von UCT ist eine abgewandelte Form der Breitensuche. Es werden dazu stets alle Kinder eines Blattknotens erstellt und mindestens einmal simuliert. Dabei werden allerdings die Knoten, die keine Blätter sind, nach ihrer bisherigen Bewertung sortiert. Dadurch entwickelt sich die Breitensuche mit dem größer werdenden Baum zu einer Best-First-Suche. Durch die Einführung von Bewertungen für unerforschte Knoten, wird die Breitensuche vollständig durch eine Best-First-Suche ersetzt. Die Bewertungen unerforschter Knoten werden von (Gelly & Wang, 2006) als First-Play-Urgencies (FPU) bezeichnet. In dieser Arbeit werden vier Ideen für FPUs gegenübergestellt.

1.2 Das Ziel der Arbeit

Die UCT-Suche steht im Mittelpunkt dieser Arbeit. Zu Beginn wird eine geeignete Repräsentation des Spiels vorgestellt. Dabei werden die Konzepte der Implementierung für alle grundlegenden Aktionen von Go beschrieben. Zu einigen Problemen werden optionale Vorgehensweisen angedacht. Die gewählte Implementierung soll für die zeitkritischen Simulationen der UCT-Suche effizient sein.

Die Baumsuche kann auf verschiedene Weisen optimiert werden. Die Arbeit beschränkt sich auf die First-Play-Urgencies. Es werden verschiedene Ansätze vorgestellt, um einen viel versprechenden, unerforschten Knoten zu finden. Die Ansätze werden miteinander verglichen. Dabei wird hervorgehoben, wie die FPUs die UCT-Suche beeinflussen.

Viele erfolgreiche Go-Programme verwenden Datenbanken aus häufigen Mustern oder enthalten bereits Lösungen für eine Vielzahl an verschiedenen Stellungen. Ein solcher Ansatz soll hier vermieden werden. Die verschiedenen, vorgestellten Varianten von UCT sollen möglichst einfach auf andere Baumsuchprobleme übertragbar sein. Ein Vorteil des Verfahrens ist die Wartbarkeit des Programms. Das verwendete Wissen über Go ist trivial und dient in erster Linie der Performance. Der Spieler wird nicht verbessert, indem ihm mehr Wissen zu Verfügung gestellt wird. Vielmehr steigt die Spielstärke mit der Leistung der ausführenden Maschine. Programme, die auf eine Wissensdatenbank zurückgreifen, können von professionellen Spielern leichter erlernt werden. Dem menschlichen Spieler ist es eventuell möglich die Schwächen des Programms gezielt auszunutzen. Auch das vorgestellte Verfahren hat erlernbare Schwächen. Die zufallsbasierte Suche ermöglicht es allerdings, überraschende Zugfolgen zu finden, die über das übliche Verhalten hinausgehen.

1.3 Der Aufbau

Die Arbeit gliedert sich in vier Teile. In Kapitel 2 wird das Spiel Go erklärt. Die wichtigsten Spielelemente und deren Implementierung werden erläutert. Kapitel 3 behandelt die Baumsuche. Der Algorithmus lässt den Spielbaum wachsen. Daher werden im ersten Abschnitt wichtige Optimierungen der Datenstruktur gezeigt. Der Schwerpunkt von Kapitel 3 liegt auf dem Suchalgorithmus. Hier wird die Monte-Carlo- und die UCT-Suche im Detail vorgestellt. Die Parametrierungen von UCT werden in Kapitel 5 untereinander verglichen. Hier liegt der Schwerpunkt auf der Auswirkung verschiedener First-Play-Urgencies. Das Kapitel schließt mit der Schlussfolgerung.

Kapitel 2

Die Implementierung von Go

Dieser Abschnitt befasst sich mit den wichtigsten Begriffen aus Go und wie diese implementiert werden können. Das Spiel baut auf wenigen Konzepte auf. In diesem Kapitel wird verdeutlicht, welche Auswirkungen ein Konzept auf den Charakter des Spiels hat.

Go ist ein Brettspiel für zwei Spieler, die abwechselnd Steine auf das Spielbrett setzen. Gesetzte Steine werden nie verschoben. Das Brett wird hauptsächlich mit 19×19 und 9×9 Kreuzungen gespielt. Es gibt einige, teils exotische Varianten des Spielbretts. Am beliebtesten ist die Änderung der Größe. Bei einer geraden Zahl an Kreuzungen verschwindet der Mittelpunkt. Es können auch Nachbarschaften zwischen Randpunkten definiert werden, so dass eine Position immer vier Nachbarn besitzt. Dadurch sind alle Spielpositionen zu Beginn gleichwertig. Abbildung 2.1 zeigt einen Spielstand in Go. Diese Arbeit beschränkt sich auf Spiele mit 9×9 Kreuzungen. Die Kreuzungen werden im Folgenden aufsteigend von links unten nach rechts oben durch Punkte der Form $(Zeile, Spalte)$ mit $Zeile, Spalte \in \{1, \ldots, 9\}$ beschrieben.

2.1 Gruppen und das Schlagen von Steinen

Gleichfarbige Steine, die horizontal oder vertikal benachbart sind, werden zu einer Gruppe zusammengefasst. Freie Kreuzungen, die an eine Gruppe angrenzen, werden Freiheiten genannt. Hat eine Gruppe keine Freiheiten, wird sie vom Brett entfernt. Setzt Weiß in Abbildung 2.1 auf $(3, 5)$, nimmt sie der schwarzen Gruppe auf $(1, 3)$ die letzte Freiheit. Weiß schlägt dadurch die schwarze Gruppe. Das Schlagen von Steinen ist eines der wichtigsten Konzepte in Go. Positiv formuliert, darf ein Spieler einen Bereich definieren, den sein Gegner mit einer bestimmten Anzahl an Zügen umschließen muss. Diese Sichtweise zeigt, dass der geschlagene Spieler durchaus einen Nutzen

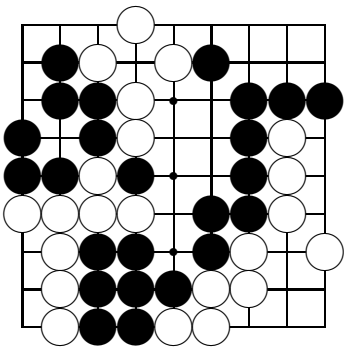

Abbildung 2.1: Eine Go-Stellung auf einem 9×9-Brett. Die Spielsteine werden auf die Gitterpunkte aus den neun horizontalen und vertikalen Linien gesetzt. Einige Kreuzungen werden speziell markiert. Die Markierungen dienen zur Orientierung und legen die Platzierung von Vorgabesteinen fest.

aus dem Verlust seiner Züge ziehen kann, da er seinen Gegner lenkt. Die Definitionen von Gruppe und Freiheiten ist vielmehr eine Definition für das Schlagen von Steinen.

Eine Implementierung der Gruppen

Gleichfarbige Steine, die horizontal oder vertikal benachbart sind, gehören zur gleichen Gruppe. Abbildung 2.2 zeigt alle Änderungen an Gruppen, die durch einen Zug entstehen können.

Datenstrukturen für Gruppen erleichtern die Suche nach geschlagenen Gruppen und das Entfernen von Steinen. Mit der Datenstruktur *disjunkte Mengen* lassen sich Gruppen gut darstellen. Eine Gruppe wird durch einen ihrer Steine repräsentiert. Der gewählte Stein enthält die Gruppeninformation. Dort wird gespeichert, wie viele Freiheiten eine Gruppe besitzt und welche gegnerischen Gruppen angrenzen. Alle Steine einer Gruppe verweisen auf ihren Repräsentanten. Dadurch lässt sich die Gruppeninformation von einem beliebigen Stein abrufen. Die Anfrage wird an den Repräsentanten delegiert. Die ersten beiden Veränderungen an Gruppen aus Abbildung 2.2 können als paarweise Vereinigung mehrerer Gruppen betrachtet werden. Für die Vereinigung wird ein Repräsentant dem Anderen untergeordnet und die Gruppeninformation aktualisiert. Es entsteht eine Baumstruktur wie in Abbildung 2.3 verdeutlicht wird. Jeder Stein kann in $O(\log n)$ die Information seiner Gruppe abrufen. Die Komplexität bezieht sich hierbei auf die Anzahl der Felder bei beliebigen Brettgrößen. Damit die Laufzeit für realistische Brettgrößen konstant wird, lässt man den Baum zusammenfallen. Wenn Informationen über eine Gruppe abgerufen werden, ersetzen alle delegierenden

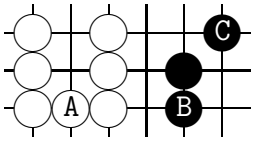

Abbildung 2.2: `A` vereinigt Gruppen. Die Freiheiten der entstehenden Gruppe sind nicht trivial berechenbar, weil Steine sich Freiheiten teilen. `B` erweitert eine Gruppe. Die Freiheiten sind einfach zu bestimmen. `C` beginnt eine neue Gruppe, deren Freiheiten gezählt werden müssen.

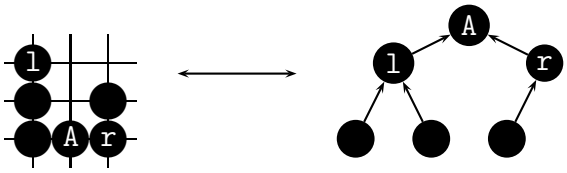

Abbildung 2.3: Nach der Vereinigung durch `A` sei der gesetzte Stein der neue Repräsentant. Die Gruppen links und rechts werden untergeordnet. Die Wahl des Repräsentanten ist hier nicht optimal.

Knoten ihren Vaterknoten durch den Repräsentanten. Dadurch verkürzen sich die Wege für folgende Anfragen.

Eine Implementierung zum Schlagen von Gruppen

Gelingt es einem Spieler, einer gegnerischen Gruppe alle Freiheiten zu nehmen, wird sie geschlagen. Sowohl die Größe als auch die Form einer Gruppe beeinflussen die Anzahl ihrer Freiheiten. Die genaue Zahl der Freiheiten lässt sich nicht immer trivial berechnen (Abbildung 2.2). Schwierigkeiten entstehen durch die Vereinigung von Gruppen mit gemeinsamen Freiheiten oder nach dem Schlagen einer Gruppe.

Für die Simulation ist es bereits ausreichend zu wissen, ob eine Gruppe mindestens eine Freiheit besitzt (lebt) oder stirbt. Die Anzahl der Freiheiten einer Gruppe lässt sich nach oben beschränken, indem man die Zahl an freien Feldern um Jeden ihrer Steine aufsummiert. Felder, die zu mehreren Steinen einer Gruppe benachbart sind, werden nun mehrfach gezählt. Die obere Schranke ist exakt, wenn eine Gruppe keine Freiheiten besitzt. Die Vereinigung von Freiheiten hat eine einfache Addition zur Folge. Für die weiße Gruppe aus Abbildung 2.2 werden nun 16 Berührungen statt 13 Freiheiten gezählt.

Zum Entfernen einer Gruppe wird eine verkettete Liste aus ihren Steinen

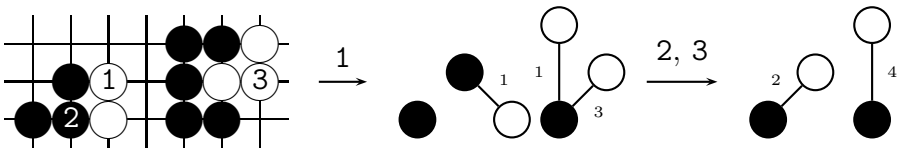

Abbildung 2.4: Rechts vom Brettausschnitt stehen die Nachbarschaftsgraphen nach dem Setzen der nummerierten Steine. Nach dem Setzen von 1 ist die linke, weiße Gruppe mit einem schwarzen Stein benachbart. Setzt Schwarz 2 kommt ein Berührungspunkt hinzu. Weiß vereinigt zwei Gruppen mit 3, die zu der gleichen, schwarzen Gruppe benachbart sind. Die Gewichte werden addiert und Mehrfachkanten gelöscht.

mitgeführt. Dadurch ist das Entfernen in $O(n)$ möglich. Es ist auch denkbar den Repräsentanten auf eine Gruppe verweisen zu lassen, die alle leeren Felder enthält. Dann gleiche das Schlagen einer Gruppe der Vereinigung mit der leeren Gruppe. Dafür wäre es schwieriger die freien Felder zu verwalten.

Beim Schlagen einer Gruppe, bekommen angrenzende, gegnerische Gruppen Freiheiten zurück. Um diese schnell zu bestimmen, wird ein Nachbarschaftsgraph benutzt. Jede Gruppe wird durch einen Knoten im Graph dargestellt. Benachbarte Knoten sind stets Gegner. Die Kanten sind mit der Anzahl der Berührungspunkte gewichtet. Dieses Gewicht lässt sich wie bei der Vereinigung befreundeter Gruppen inkrementell aktualisieren. Konfrontiert ein gesetzter Stein gegnerische Gruppen, wird eine Kante erstellt. Besteht bereits eine Kante werden die Berührungspunkte mit dem gesetzten Stein zu ihrem Gewicht addiert. Ein Nachbarschaftsgraph ist in Abbildung 2.4 dargestellt. Der Knoten für eine geschlagene Gruppe wird aus dem Graphen entfernt und die Kantengewichte den Nachbarn als Freiheiten zurückgegeben.

2.2 Verbotene Züge

Die Spieler dürfen im Wesentlichen auf alle freien Kreuzungen setzen. Hierdurch entstehen viele Zugvarianten, die es schwierig machen eine gute Strategie abzuschätzen. Schon durch Veränderung weniger Züge kann das Spiel einen neuen Charakter bekommen. Diese Vielfalt vor allem in der frühen Phase des Spiels macht es schwierig eine Stellung zu bewerten. In wenigen Fällen ist es allerdings verboten oder zumindest falsch auf Kreuzungen zu setzen.

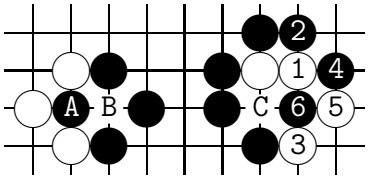

Abbildung 2.5: Weiß kann auf `B` setzen, um den Stein auf `A` zu schlagen und erzeugt dadurch Ko. Für Schwarz ist es verboten direkt wieder auf `A` zu spielen. Dieser Zug würde die abgebildete Situation wiederherstellen. Setzt Weiß allerdings auf `C` entsteht keine Ko-Situation. Schwarz kann sofort zurückschlagen und erzeugt dadurch einen neuen Zustand.

2.2.1 Die Ko-Situation

Im Laufe des Spiels kann eine Stellung entstehen, in der es möglich ist einen früheren Spielstand wiederherzustellen. Diese Situation nennt man Ko. Der Zug der eine vorherige Brettkonfiguration erzeugt ist verboten. Große Zyklen treten zwar selten auf, können aber nur mit teurem Speicherplatzverbrauch erkannt werden. In langen Zyklen müssen viele Steine geschlagen werden. Durch eine obere Schranke an die Gesamtzahl der gefangenen Steine, kann man abschätzen, wann der Spieler von einem Zyklus ausgehen kann. Häufiger sind allerdings Zyklen aus zwei Zügen. Ein solcher Zyklus kann nur entstehen, wenn die Spieler um einen Stein kämpfen. Eine solche Situation zeigt Abbildung 2.5. Um triviale Zyklen aus zwei Zügen zu verhindern, kann man den Zug auf eine Kreuzung verbieten, von der ein einzelner Stein entfernt wurde. Abbildung 2.5 zeigt allerdings auch, dass eine Kreuzung nur dann gesperrt werden darf, wenn der schlagende Stein eine neue Gruppe erzeugt. Um Ko automatisch festzustellen reicht es also die Veränderung der Gefangenen und die Größe der Gruppe des neuen Steins zu beobachten. Diese Definition von Ko ist nicht exakt, aber dennoch hinreichend. Es wird fälschlicher Weise Ko erkannt, wenn die Kreuzung `B` in Abbildung 2.5 nicht von schwarzen Steinen umschlossen ist. In den folgenden Abschnitten wird jedoch deutlich, weshalb ein Zug auf die Kreuzung dennoch verboten werden darf.

Ko wird für jeden Zustand des Spiels neu entschieden. Aus Spiel entscheidenden Ko-Stellungen entstehen wichtige, strategische Elemente in Go. Die strategische Bedeutung von Ko wird auf (Sensei's Library, 2000a) beschrieben.

2.2.2 Selbstmord

Ein Spieler begeht Selbstmord, wenn er eine eigene Gruppe schlägt. Abbildung 2.5 zeigt, dass ein Stein in manchen Situationen entweder die eigene oder eine gegnerische Gruppe schlagen könnte (der Zug auf `C`). In diesem Fall wird die Gruppe des Gegners geschlagen. In manchen Regeln ist Selbstmord erlaubt - In den Spielen dieser Arbeit allerdings nicht. Selbstmord kann vor allem als Drohung Sinn machen. Näheres dazu beschreibt (Sensei's Library, 2000b).

Ein Zug ist niemals Selbstmord, wenn der gesetzte Stein Freiheiten hat. Andernfalls ist die Entscheidung aufwendiger. Für jede Gruppe muss entschieden werden, ob sie durch den Zug sterben könnte oder überlebt. Eine freie Kreuzung wird genau dann nicht gesperrt, wenn eine befreundete, benachbarte Gruppe den Zug überlebt oder eine gegnerische Gruppe geschlagen wird. Statt Freiheiten werden wieder die Berührungen einer Gruppe mit der zu prüfenden Kreuzung gezählt. Eine überlebende Gruppe, hat mehr freie Berührungspunkte als ihr durch den Zug genommen werden können.

2.2.3 Lebende Gruppen haben Augen

Um eine Gruppe zu schlagen, müssen alle Freiheiten von gegnerischen Steinen besetzt werden. Demzufolge kann eine Gruppe nicht mehr geschlagen werden, wenn der Gegner dazu mindestens zwei Züge bräuchte und diese Selbstmord bedeuten. Dies gilt auch, wenn Selbstmord erlaubt ist. Freie Felder, die einen Spieler zum Selbstmord zwingen heißen Augen. In Go geht es also darum, möglichst große Gruppen mit mindestens zwei Augen herzustellen.

Der in dieser Arbeit verwendete Lernalgorithmus, ist in der Lage das Konzept von Augen zu erlernen. Der Algorithmus erzeugt dafür zufällige Spiele. Ohne das Wissen über Augen dauert es jedoch lange, bis ein Spiel endet. Gruppen sind selten stabil und werden häufig geschlagen. Um den Lernprozess zu beschleunigen ist es deshalb sinnvoll primitive Patterns vorzugeben. Anhand diesen, verbietet sich der Spieler, Augen zu füllen. Eine frühe Überlegung für die Implementierung war, einem Spieler zu verbieten einen Zug zu spielen, der einer eigenen Gruppe nur noch eine Freiheit lässt. Mit dieser Regel werden mehr Fälle abgedeckt, als nötig sind, um Augen zu schützen. Es scheint zunächst vernünftig Gruppen nicht mit einer Freiheit zurück zulassen, weil es dem Gegner direkt möglich wäre diese Gruppen zu schlagen. In Abbildung 2.5 finden sich allerdings kritische Beispiele. Ein Ko wird nie geschlagen, weil der gesetzte Stein ein Gruppe mit nur einer Freiheit bildet. Der sechste Zug aus der Abbildung zum Fangen der weißen Gruppe ist nicht möglich. Das Opfern von Steinen ist ein wichtiges Element von Go

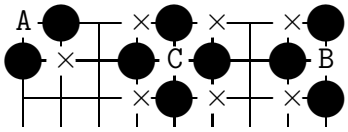

Abbildung 2.6: Augen werden durch die folgenden, drei Patterns erkannt. `A` und `B` gelten beide solange als Auge, bis ein gegnerischer Stein auf ein × gesetzt wird. `C` bleibt ein Auge, solange sich höchstens ein gegnerischer Stein auf den umliegenden, markierten Kreuzungen befindet. Umliegende Steine der gleichen Farbe werden vom Pattern ignoriert, weil sie das Auge nicht gefährden.

und kann über Leben und Tod von Gruppen beider Spieler entscheiden.

Die strategischen Einschränkungen erfordern daher eine alternative Erkennung. Die implementierten Patterns zeigt Abbildung 2.6. Die dargestellten Patterns beurteilen ein Auge nur anhand der umliegenden Kreuzungen. Sie haben die Aufgabe, abzuschätzen, wann eine Kreuzung in Gefahr sein könnte. Ecken und Kanten geraten schnell in Bedrängnis. Sobald ein gegnerischer Stein auftaucht, ist es möglich, dass ein Stein am Rand geschlagen werden kann. Der Spieler sollte das Auge überdenken. Bei Augen, die nicht zum Rand benachbart sind, ist vor allem der Fall, dass zwei, gegnerische Steine diagonal angreifen, interessant. Das Pattern ist hier sehr kritisch. Es nimmt an, dass die schwarzen Steine aus Abbildung 2.6 zu zwei Gruppen gehören. Von den Gruppen könnte mindestens eine von gegnerischen, weißen Steinen umzingelt sein. Daher ist das Auge möglicherweise in Gefahr und muss überdacht werden.

Die drei Patterns aus Abbildung 2.6 basieren auf der Augenerkennung von (Brügmann, 1993). Die Erkennung am Rand wird durch die vorgestellten Patterns sensibilisiert. In (Brügmann, 1993) sind zwei Steine am Rand nötig, um ein Auge zu gefährden. Die Ecksituation ist nie in Gefahr. Es lassen sich leicht Szenarien konstruieren, in denen eine Position nicht verteidigt wird, weil der Spieler, durch die optimistische Erkennung glaubt, ein eigenes Auge zu füllen.

2.3 Das Spielende

Ein Spieler kann entweder einen Stein setzen oder passen. Passen bedeutet, dass er einen Zug aussetzt und, sofern sein Gegner einen Stein setzt, in der folgenden Runde wieder an der Reihe ist. Passen beide Spieler hintereinander, endet das Spiel, weil offensichtlich niemand einen entscheidenden Zug

spielen kann. Dies hat hier zur Folge, dass beide Spieler solange setzen, bis kein legaler Zug mehr möglich ist. Man kann erkennen, dass alle vorgestellten Zugverbote immer nur eine freie Kreuzung betreffen. Sobald zwei freie Felder benachbart sind, gibt es somit mindestens einen legalen Zug. Zusätzlich zu den Go-Regeln, wird das Spiel abgebrochen, wenn die Anzahl der geschlagenen Steine einen Grenzwert überschreiten. Dieses Kriterium ist eine Heuristik für eine Zyklenerkennung, die über das bereits vorgestellte Ko-Pattern hinausgeht.

Der Gewinner einer Partie wird in Go anhand der Gebietspunkte festgestellt. Gebiete sind freie Felder, die von Steinen eines Spielers begrenzt sind. Es gibt verschiedene Auswertungen, in denen die Randsteine entweder mitgezählt oder ignoriert werden. Die chinesische Zählweise ist einfach zu implementieren. Die Punkte eines Spielers setzen sich aus der Anzahl seiner Steine auf dem Spielfeld und jeder freien, von ihm umschlossenen Kreuzung zusammen. Er bekommt jeweils einen Punkt. Die Zählweise lässt sich weiter vereinfachen, weil ein beendetes, simuliertes Spiel keine benachbarten, freien Kreuzungen enthält. Es muss lediglich getestet werden, wessen Auge eine freie Kreuzung ist.

Kapitel 3

Die Suche im Spielbaum

Die Struktur eines Spiels lässt sich als Graph oder auch als Baum modellieren. Dabei ist ein Spielzustand von einem Anderen erreichbar, wenn es eine gültige Zugfolge bezüglich der Regeln gibt. In dieser Struktur lassen sich nun Strategien der Spieler untersuchen. Suchalgorithmen suchen theoretisch nach einer Folge von Aktionen, die für alle Spieler optimal ist. Das bedeutet, dass keiner der Spieler eine Aktion ändern möchte, weil er dadurch seine Erwartung nur verschlechtern kann (Nash, 1950). In der Praxis ist eine erschöpfende Suche nicht möglich. Daher lohnt es sich Algorithmen zu betrachten, die gegen eine optimale Strategie konvergieren. Dabei soll die Qualität der Zwischenlösungen mit der Rechenzeit zunehmen. Der verwendete Algorithmus UCT erfüllt diese Bedingungen.

3.1 Der Spielbaum

Ein Spielgraph modelliert die Stellungen und Zustandsübergänge. Da in vielen Spielen verschiedene Zugfolgen zur gleichen Stellung führen, ergibt sich ein Graph. Im Fall von Go enthält dieser auch Zyklen. Im Folgenden wird Go allerdings mit einem Spielbaum modelliert. Gegen einen Graphen sprechen zwei Argumente. Die Knoten werden erzeugt, wenn sie zum ersten Mal erkundet werden. Es muss also bestimmt werden, ob ein Knoten bereits besteht oder noch erstellt werden muss. Zum effizienten Suchen nach Stellungen können Schlüssel nach dem Verfahren von Zobrist generiert werden (Zobrist, 1970). Kollisionen der Hashkeys sind unwahrscheinlich aber nicht ausgeschlossen. (Gelly & Wang, 2006) bemerken, dass die Redundanz im Baum im Vergleich zu anderen Problemen in Go vernachlässigbar ist. Der Graph ist nur schwach zusammenhängend. Eine Breitensuche vom eindeutigen Knoten mit Eingangsgrad Null ergibt mehr Vorwärts- als Rückwärtskanten. Daher

wird Go als Baum modelliert.

Die Wurzel des Baums repräsentiert das leere Brett. Auf der k-ten Ebene befinden sich alle Spielstellungen, die mit k Zügen generiert werden können. Die Blätter des vollständigen Spielbaums sind Endstellungen. Da der Spielbaum iterativ wächst, werden mit Blättern im Folgenden die letzten Knoten des unvollständigen Baums bezeichnet. Der Verzweigungsgrad eines Knotens ist durch die Anzahl der möglichen Züge beschränkt und somit endlich. Die Tiefe des Baums ist dagegen für einige Zugfolgen unendlich. Diese Zugfolgen sind Zyklen im Spielgraph.

Ein Knoten im Baum repräsentiert eine Stellung in Go. Aufgrund der Baumstruktur können in verschiedene Knoten gleiche Spielzustände auftreten. Es ist allerdings aus Speicherplatzgründen zu teuer das Brett mit allen Steinen in jedem Knoten zu speichern. Der Spielstand wird nur in Knoten benötigt, die vom Algorithmus untersucht werden. Die verbleibenden, gültigen Züge werden aus dem Spielstand ermittelt. Es ist daher überflüssig Stellungen in Knoten zu speichern, deren Kinder vollständig erstellt wurden. Das Löschen überflüssiger Spielzustände bringt allerdings keinen deutlichen Gewinn. Die Intention des verwendeten Algorithmus ist es schnellstmöglich nach der besten Strategie zu suchen. Die Suche in die Tiefe erzeugt somit hauptsächlich Knoten, die noch nicht vollständig expandiert wurden. Daher ist es sinnvoll, generell auf das Speichern der Bretter zu verzichten. Statt dessen wird nur der letzte Zug zum Knoten vermerkt. Werden Knoten entlang eines Pfads gewählt, wird der Zustand iterativ, immer wieder neu aus der Stellung der Wurzel berechnet. Durch die zusätzliche Berechnung verliert der Spieler nur wenig Zeit.

Der Algorithmus UCT verwendet Simulationen. Wird ein Knoten simuliert, muss ohnehin ein Spiel zufällig generiert werden. Die Tiefe des unvollständigen Spielbaums, in den der Algorithmus absteigt, ist im Schnitt nur ein Bruchteil der Züge in der darauf folgenden Simulation.

Entscheidet sich der Algorithmus endgültig für einen Zug, der dann gespielt wird, müssen alle Alternativen zum Gewählten nicht mehr betrachtet werden. Der gewählte Knoten wird zur neuen Wurzel. Alle folgenden Spielstände können aus der Stellung des gewählten Knotens berechnet werden. Es ist daher nicht nötig die Spiele vom leeren Spielfeld zu entwickeln. Beim Abschneiden von nicht mehr benötigten Zugvarianten verbleiben nur die simulierten Knoten, die noch für das restliche Spiel nützlich sind. Im besten Fall wurden die verbleibenden Knoten bereits ausführlich untersucht. Damit ist es dem Algorithmus möglich effektiv weitere, gute Züge zu finden. Andernfalls muss die Suche eventuell eine neue Strategie erzeugen. In diesem Fall ist zu hoffen, dass der Gegner einen möglichst schlechten Zug gewählt hat. Es ist dann schnell möglich eine bessere Strategie zu finden.

3.2 Suchalgorithmen

Der Algorithmus UCT, der in dieser Arbeit verwendet wurde, benutzt zufällige Simulationen. Ziel ist es statistische Messungen zu erheben, aus denen man auf die Güte eines Zugs schließen kann. Bei derartigen Suchproblemen muss man abwägen, ob man einen derzeitig guten oder einen unsicheren Zug untersucht. Untersucht man den besten Zug, der bislang gefunden wurde, ist es eventuell möglich weiterführende Strategien zu planen. Dafür erfährt man nur wenig über die anderen Züge, unter denen unter Umständen ein noch viel besserer Zug zu finden ist. Entscheidet man sich, die Züge möglichst homogen zu untersuchen, erfährt man mehr über alternative Züge. Dafür ist der beste, gefundene Wert unsicherer. Im schlimmsten Fall wird der beste Zug aufgrund einer zu hohen Unsicherheit nicht entdeckt. Dieses Problem wird als das Exploration-Exploitation-Dilemma bezeichnet.

3.2.1 Monte-Carlo-Simulation

Monte-Carlo-Methoden approximieren die Lösung einer Optimierung. Dabei werden statistische Experimente durchgeführt. Eine offensichtliche Anwendung für die Suche nach einer optimalen Strategie findet sich in Spielen mit unvollständiger Information wie Poker oder Backgammon. Hier dient die Simulation im Wesentlichen dazu, den wahrscheinlichsten Zustand der fehlenden Informationen zu modellieren. Das Go spielende Programm *Gobble* aus (Brügmann, 1993) zeigt, dass die Monte-Carlo-Simulation auch erfolgreich in Go angewendet werden kann. Der Erfolg des Programms liegt darin, dass es möglich war einen soliden Go Spieler zu erstellen, dem nahezu kein Expertenwissen zur Verfügung stand.

Gobble setzt den Grundstein für viele Simulationsansätze in Go. Die Monte-Carlo-Suche sucht nach dem wahrscheinlich besten Zug. Ein beliebiger, gültiger Zug wird ausgewählt. Zu diesem wird ein zufälliges, vollständiges Spiel generiert. Gewinnt der Spieler dieses Spiel, verbessert sich die Bewertung des Zugs. Nachdem alle Zugvarianten häufig simuliert wurden, wählt der Algorithmus den Zug mit der besten Bewertung und spielt diesen. *Gobble* verwendet zusätzlich simulated Annealing, um einen Zug in der Zugliste zu verschieben. Der Auskühlungsprozess des simulated Annealings dient dazu, gegen Ende häufiger den derzeitig besten Zug an den Anfang der Liste zu binden und balanciert dadurch das Exploration-Exploitation-Problem. Die allgemeine Monte-Carlo-Suche nach dem wahrscheinlich besten Zug wird in Algorithmus 1 formuliert. SELECT wählt ein zufälliges Kind der Wurzel.

Die Bewertung eines Zugs ist der Mittelwert über die simulierten Spielauswertungen. Die Stabilität des Algorithmus wird durch die Wahl der Spiel-

Algorithm 1 MC(*root*, *threshold*)

1: **for** *step* := 1 to *threshold* **do**
2: *node* := SELECT(*root*)
3: *value* := SIMULATE(*node*)
4: UPDATE(*node*, *value*)
5: **end for**
6: **return** $\arg\max_v\{v.value \mid (root, v) \in E\}$

auswertung beeinflusst (Gelly & Wang, 2006). Vor allem für das UCB1-Verfahren, dass im folgenden Abschnitt zum UCT-Algorithmus führt, müssen die Erträge einer Simulation zwischen 0 und 1 liegen.

Gebietswertung

Der Ausgang eines Spiels bestimmt sich normalerweise aus der Gebietswertung. Die Gebietswertung wird für die vorgestellten Implementierungen mit der chinesischen Zählweise bestimmt. Da die Erträge aus $[0, 1]$ sein müssen, muss man zusätzlich mit der Maximalpunktzahl normieren. Die erreichbare Punktzahl in Go ist durch die Brettgröße beschränkt. Die Erträge eines Spiels werden häufig in der Nähe von 0.5 liegen. Viele, zufällig gespielte Endstellungen tragen nur geringfügig zur Unterscheidung zweier Züge bei. Es werden viele Simulationen benötigt, um einen eindeutigen, besten Zug abzugrenzen. Dazu muss beachtet werden, dass die Spiele zufällig ausgespielt werden. Erst gegen Ende der tatsächlichen Partie sind die simulierten Gebiete realistisch. Zu Beginn kann die erwartete Gebietsgröße nicht bestimmt werden.

0/1-Wertung

Die Endstellung wird nach der chinesischen Wertung ausgezählt. Gewinnt der Spieler, wird der simulierte Zug mit einer 1 bewertet. Ansonsten bekommt er eine 0. Große Differenzen in der Spielauswertung fallen nun nicht mehr ins Gewicht. Lediglich die Anzahl der gewonnen Spiele ist relevant. Der Mittelwert gibt nach hinreichend vielen Simulationen die Gewinnwahrscheinlichkeit an. Die 0/1-Wertung ist robust und wird im Folgenden für die Bewertung eines Zugs herangezogen.

Weitere Bewertungsfunktionen

Im Laufe der Arbeit wurden alternative Bewertungsfunktionen ausprobiert. Die Motivation und Beobachtungen sollen hier kurz erläutert werden.

Kleine Differenzen Zunächst wird die Punktedifferenz der chinesischen Zählung ermittelt. Die 0/1-Wertung wird mit der inversen Punktedifferenz gewichtet. Ausgewogene Spiele bekommen dadurch ein stärkeres Gewicht. Weil solche Spiele jedoch seltener auftreten, stellt sich ein guter Knoten nur langsam heraus. Es wird vorwiegend eine Strategie verfolgt, bei der beide Spieler gleich schlecht spielen.

Wenige Gefangene Spiele mit großer Punktedifferenz haben häufig eine hohe Anzahl geschlagener Steine. Das Spiel wird chinesisch ausgezählt. Die Punkte eines Spielers werden mit dem Kehrwert der von ihm gefangenen Steine gewichtet. Eine hohe Bewertung deutet darauf hin, dass ein Spieler durch geschickte Züge Gebiet erschlossen hat. Diese Bewertung bewirkt jedoch ebenfalls, dass der Spieler generell ungern Gruppen schlägt.

Kurze Spiele Die 0/1-Wertung wird mit der inversen Spieldauer multipliziert. Da viele kurze Spiele noch nicht beendet sein können, ist auch eine Sigmoid-Funktion denkbar. Durch diese lässt sich präziser Steuern, wann die Bewertungen sinken. Ein kurzes Spiel beinhaltet, dass selten große Gruppen geschlagen werden. Es werden wenige, überflüssige Züge gespielt. Außerdem kann man die Simulation abbrechen, wenn die Bewertung des Spiels unter einen Grenzwert nahe bei Null fällt. Diese Bewertung hat sich nicht gegen die 0/1-Wertung durchsetzen können.

Die Gewichtungen haben sich nicht ausreichend auf das Spielverhalten ausgewirkt. Daher wurde der Einsatz von alternativen Bewertungsfunktionen verworfen. Die Begrenzung durch die Spieldauer scheint jedoch der aussichtsreichste Ansatz zu sein. Vor allem der frühe Abbruch von Simulationen wurde weiterverfolgt, um nicht-triviale Ko-Zyklen zu vermeiden und die Dauer einer Simulation zu beschränken.

3.2.2 UCT Simulation

Das Exploration-Exploitation-Dilemma findet sich im Multiarmed-Bandit-Problem wieder. Verschiedene Banditen sind unterschiedlich zuverlässig. Der Wert der Beute variiert nach einem erledigten Auftrag. Die Aufgabe ist es, einen Banditen zu finden, der den Ertrag maximiert. Um dieses Problem zu lösen, muss man einen Kompromiss zwischen Erkunden und dem Ausschöpfen guter Züge finden. Ziel des Kompromisses ist es, das Simulieren von nicht optimalen Zügen am wenigsten zu bereuen. Die zu minimierende

Reue wird definiert als

$$\text{regret} := n\mu^* - \sum_{i=1}^{K} \mathbb{E}(n_i)\mu_i.$$

μ^* ist der maximale Ertrag. n bezeichnet die Anzahl der Aufträge. $\mathbb{E}(n_i)$ ist die erwartete Häufigkeit, mit welcher der i-te Bandit beauftragt wird. Das Produkt $n\mu^*$ ist der Ertrag, den man erhält, wenn stets der optimale Bandit beauftragt wird. Da andere Banditen losgeschickt werden, entsteht ein Verlust. Lai und Robbins haben für ausgewählte Ertragsverteilungen eine obere Schranke für $\mathbb{E}(n_i)$ gefunden. Sie haben außerdem gezeigt, dass der Verlust für diese Schranke minimal ist (Auer, Cesa-Bianchi, & Fischer, 2002).

(Auer et al., 2002) haben auf dieser Schranke UCB1 (Upper Confidence Bound) entwickelt. Dieses Verfahren funktioniert für alle Ertragsverteilungen, solang diese durch $[0, 1]$ beschränkt sind. Insbesondere können die Erträge sowohl zwischen den einzelnen Banditen als auch zwischen den Aufträgen unabhängig verteilt sein.

UCT (Upper Confidence bound applied to Trees) erweitert die Monte-Carlo-Suche und integriert den UCB1-Kompromiss, um zwischen den Zügen zu wählen. Der Algorithmus sucht nach der wahrscheinlich besten Strategie. Es wird eine Folge von Zügen gesucht, die für beide Spieler möglichst gut ist. Der Algorithmus ist in vier Abschnitte unterteilt.

Das Absteigen

Mit einer Best-First-Suche steigt der Algorithmus in den Spielbaum ab. Jeder Knoten wird separat als Multiarmed-Bandit-Problem betrachtet. Ausgehende Kanten stehen für legale Züge in die Brettstellung des Knotens. Für jeden Pfad im Spielbaum lässt sich ein resultierender Spielstand konstruieren. Dazu werden die Züge auf dem Pfad nacheinander in die Ausgangsstellung gespielt. In jeder Iteration des UCT-Algorithmus wird der Baum um einen Knoten erweitert. Die Knoten werden anhand eines UCT-Werts ausgewählt. Der UCT-Wert gleicht dem UCB1-Verfahren und ist für Bäume definiert als

$$\text{UCT-Value}(v) := \bar{x}_v + c \cdot \sqrt{\Delta_v \cdot \min\{1/4, \sigma_v^2 + \sqrt{2\Delta_v}\}}, \qquad (3.1)$$
$$\Delta_v := \frac{\ln n_{\text{parent}(v)}}{n_v}.$$

$\bar{x}_v$ bezeichnet die Bewertung des Knotens v. Sie wird durch Monte-Carlo-Simulationen iterativ ermittelt. Zur Bewertung kommt der „Neugierigkeitssummand“. Der Summand versucht das Exploration-Exploitation-Dilemma

zu balancieren. Δ_v beschreibt, wie häufig ein Knoten im Verhältnis zu seinen Geschwistern ausgewählt wird. Da der lineare Nenner schneller wächst als der Logarithmus im Zähler, nimmt der Wert ab, wenn v untersucht wird. v wird uninteressanter und die Relevanz seiner Geschwister steigt. Ein weiterer Faktor ist die Varianz σ_v^2. Knoten mit weit gestreuten Ergebnissen müssen häufiger simuliert werden. Der „Neugierigkeitssummand“ wächst proportional mit der Varianz, die mit $1/4$ nach oben beschränkt wird. c gewichtet die Neugierde. Je größer c gewählt wird, desto mehr erforscht der Algorithmus verschiedene Knoten. Ein kleines c veranlasst den Algorithmus intensiver den Knoten mit der bisher größten Ausbeute zu untersuchen.

Das Expandieren

In jeder Iteration wird ein neuer Knoten im Spielbaum erstellt. Bisher steigt die Best-First-Suche bis in die Blätter ab. Dies würde bedeuten, dass nur eine Zugfolge untersucht wird. Dieses Problem wird gelöst, indem jedes Kind, dass nicht erkundet wurde, einen Wert zugewiesen bekommt. Dieser Wert wird First Play Urgency genannt (Gelly & Wang, 2006). Die FPUs werden im Abschnitt 3.3 ausführlich beschrieben. Die Best-First-Suche kann dadurch auch neue Kinder auswählen. Ein neues Kind wird zufällig erstellt, wenn es mehrere gleiche First-Play-Urgencies gibt. Anstatt First-Play-Urgencies zu benutzen, ist es auch möglich alle Kinder eines Blatts zu expandieren. Jedes Kind wird daraufhin einmal simuliert und erhält dadurch seinen initialen Wert. So ist es für das UCB1-Verfahren vorgesehen. Der Vorteil der First-Play-Urgency ist, dass neue Knoten nur dann erzeugt werden müssen, wenn sie von der Best-First-Suche ausgewählt werden. Dadurch wird erheblich Speicherplatz gespart (Coulom, 2006). Außerdem kosten die Simulationen aller, neuen Kinder Zeit. (Gelly & Wang, 2006) stellen fest, dass bestimmte First-Play-Urgencies die UCT-Suche verbessern. Abschnitt 3.3 stellt verschiedene First-Play-Urgency-Funktionen FPU(v) vor.

Durch das Absteigen in Unterbäume und das Expandieren des Spielbaums ergibt sich für UCT eine abgewandelte SELECT-Methode. Die Methode aus Algorithmus 1 wird durch Algorithmus 2 ersetzt.

Die Simulation

Der expandierte Knoten wird nach dem Monte-Carlo-Ansatz einmal, oder mehrmals, zufällig simuliert. Mehrmals zu simulieren, kostet Zeit. Dafür erhält man ein stabileres Ergebnis für die aktuelle Strategie. Im Normalfall müsste die Strategie gut genug sein, um vom Selektionsschritt mehrmals ausgewählt zu werden. In dieser Arbeit wird ein ausgewähltes Blatt genau

Algorithm 2 SELECT($node$)

```
1: if node is a leaf then
2:    create new node v
3:    add v to V and (node, v) to E
4:    return v
5: end if
6:
7: next := arg max_v{UCT-Value(v) | (node, v) ∈ E}
8: fpu := max_v{FPU(v) | (node, v) ∉ E}
9: if UCT-Value(next) < fpu then
10:   create new node v
11:   add v to V and (node, v) to E
12:   return v
13: end if
14:
15: return SELECT(next)
```

einmal simuliert. Für eine Simulation wird der Spielzustand des Blattknotens kopiert und zufällig zu ende gespielt. Die Simulation erzeugt keine neuen Knoten im Spielbaum. Wichtig ist nur das Spielergebnis, welches die Strategie bewertet.

Die Werte anpassen

Das Simulationsergebnis wird vom neuen Blattknoten bis in die Wurzel hoch gereicht. Der Blattknoten ist der bisher letzte Zustand der gewählten Zugsequenz. Jeder Knoten in der Zugsequenz wird aktualisiert. Die Knoten sind dabei abwechselnd Schwarz und Weiß zugeordnet. Da die 0/1-Wertung benutzt wird, bekommen nur die Knoten des Gewinners eine 1 eingetragen. Zusätzlich ändert sich der „Neugierigkeitssummand“ für alle Geschwisterknoten der verfolgten Strategie. Da diese nicht untersucht wurden, steigt deren Relevanz. Das Simulationsergebnis wird nach Algorithmus 3 in die Wurzel gereicht. Die UPDATE-Methode aus Algorithmus 1 muss ersetzt werden.

Wenn alle Knoten aktualisiert worden sind, werden die vier Schritte erneut ausgeführt bis ein Abbruchkriterium erreicht wird. Abbruchkriterien sind zum Beispiel eine Zeitgrenze oder maximale Zahl an Iterationen.

Algorithm 3 UPDATE(*node*, *value*)

```
1: add value to node.x̄
2: node.n := node.n + 1
3:
4: if node is not the root then
5:    UPDATE(PARENT(node), 1 − value)
6: end if
```

3.3 Die First-Play-Urgency

Der Spielbaum von Go zeichnet sich dadurch aus, dass seine Knoten einen hohen Verzweigungsgrad haben. Damit ein Zug vom Algorithmus gewählt werden kann, muss für diesen ein Schätzwert existieren. Wenn ein Blatt des iterativ wachsenden Baums erreicht wird, kann man dessen Kinder expandieren und daraufhin initial simulieren. Somit berechnet man für jedes Kind den UCT-Wert und kann aufgrund dessen die Auswahl treffen. Dieses Vorgehen hat den Nachteil, dass zum Einen durch das Aufzählen sämtlicher Zugalternativen Speicherplatz verschwendet wird und zum Anderen jeder Zug durch die Auswahlheuristik irgendwann fordert, genauer erkundet zu werden.

Dieses Problem lösen First-Play-Urgencies (FPU). Die FPU ist eine Bewertung für nicht erforschte Knoten. Der Wert der FPU wird mit dem UCT-Wert simulierter Knoten verglichen. So kann der Algorithmus entscheiden, ob er den besten Zug der bereits erkundeten oder den aussichtsreichsten Zug der noch nicht erforschten Knoten wählen möchte. In den nächsten Abschnitten werden die FPUs vorgestellt, die in dieser Arbeit gegeneinander getestet werden.

3.3.1 Konstante FPU

Die einfachste Lösung für FPUs ist einen konstanten Wert für unerforschte Knoten festzulegen (Gelly & Wang, 2006). Der Wert legt fest, wie dringend es ist, einen zufälligen, neuen Zug zu wählen. Wird der Wert groß gewählt (zum Beispiel 10000), kann sich kein UCT-Wert gegen die FPU durchsetzen. Es werden dadurch alle Zugalternativen erzeugt, bevor ein Kind des Knotens ein weiteres Mal simuliert wird. Ist die FPU kleiner als Null, folgt der Spieler einer zufälligen Strategie. Eine ausgewogene Konstante interpoliert zwischen der Breiten- und Tiefensuche. Es wird kein neuer Zug expandiert, bis die UCT-Werte der erkundeten Knoten unter den festgelegten Grenzwert fallen. (Gelly & Wang, 2006) haben für MoGo den Wert 1.0 ermittelt. In den Ergebnissen dieser Arbeit stellt sich heraus, dass für die hier gewählte

Implementierung der Wert 1.4 besser geeignet ist.

3.3.2 Simulationswerte vererben

Die folgende First-Play-Urgency ist eng verwandt mit der History Heuristik (Schaeffer, 1989). Für eine α-β-Suche kann die History Heuristik zum Vorsortieren der Züge benutzt werden. Dadurch werden schnell gute Fenster für das Pruning gefunden. Als FPU wird nach Zügen gesucht, die wahrscheinlich lange Zeit alternative Züge dominieren. Go ist ein Spiel für zwei Spieler. Solange keine großen Gruppen geschlagen werden, ändert sich die Stellung zwischen zwei Zügen eines Spielers nur geringfügig. Hierin liegt hauptsächlich die Motivation für die History Heuristik. Da der Spielbaum Top-Down entwickelt wird, liegen bereits einige Simulationswerte der vorherigen Züge vor. Diese Werte können eine hilfreiche Abschätzung für folgende Züge darstellen. Soll ein neuer Zug ausprobiert werden, wird dieser nun also nicht mehr zufällig aus den nicht erforschten Zügen gewählt. Es wird eine Zugvariante vorgezogen, die sich in vorherigen Varianten bereits als sinnvoll erwiesen hat. Wie in Abbildung 3.1 gezeigt, werden nur frühere Simulationswerte des gleichen Spielers betrachtet. Würden auch Ergebnisse des Gegners betrachtet werden, könnte es passieren, dass ein Zug probiert wird, der den Spieler gefährdet.

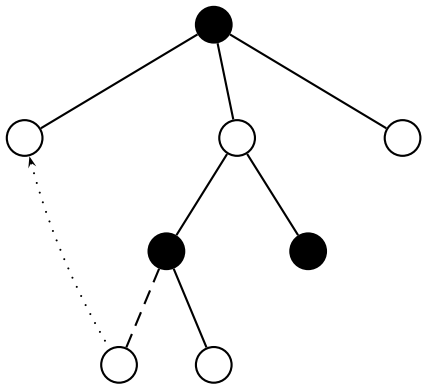

Abbildung 3.1: Der Algorithmus prüft, ob ein neuer Zug von Weiß probiert werden sollte. Er bezieht dazu den Simulationswert des entsprechenden Zugs von Weiß einer früheren Ebene. Der Simulationswert wird an die FPU nachfolgender Knoten vererbt.

Es kann vorkommen, dass es nicht ausreicht zwei Züge zurück zu schauen, weil auch dort die gesuchte Variante noch nicht probiert wurde. In diesem Fall ist es möglich eine konstante FPU wie im Abschnitt zuvor zu definieren. Man kann allerdings beobachten, dass diese Lösung das Problem nur

ungenügend behebt. Sobald eine Ebene im Spielbaum von vorangegangenen Simulationen profitieren kann, werden in dieser nur wenige Zugvarianten untersucht. Dadurch ist bereits die nächste Ebene des entsprechenden Spielers gezwungen, mit der gewählten Konstanten zufällig nach neuen Zügen zu suchen. Daher ist es sinnvoll mehr als nur zwei Züge zurück zu schauen. Es können alle Simulationswerte vererbt werden, die den gesuchten Zug erkundet haben. Gewählt wird die Referenz, die auf dem Weg zur Wurzel als Erste gefunden wird. Je ferner die Referenz dem Knoten ist, für den die FPU berechnet werden soll, desto ungenauer ist der Schätzwert. Übertragen auf die History Heuristik, ergibt der Abstand eines Knotens zum aktuellen Spielzustand und dessen Simulationswert das Gewicht für den Zug, der von der Heuristik vorgeschlagen wird.

Der Wert des Referenzknotens hat hohes Gewicht. Daher ist ein stabiler Wert eine wichtige Voraussetzung für eine sinnvolle Vererbung. Es reicht nicht aus, die Referenz nur einmal simuliert zu haben. Deshalb sollte ein Referenzknoten nur dann akzeptiert werden, wenn die Anzahl der Simulationen über einen gewissen Grenzwert liegen. Diese Schranke bremst die Suche in die Tiefe ein wenig. Allerdings verbessert sich im Idealfall die Qualität der Suche. Somit wird deutlich, dass lediglich ein Mittelweg gefunden werden muss. Im Laufe des Spiels verliert der Grenzwert an Bedeutung, weil die Unterbäume bereits Simulationen erfahren, bevor sie tatsächlich vom Spieler gewählt werden. Aufgrund des Grenzwertes wird jedoch deutlich, weshalb gegnerische Züge keine Schätzwerte vererben dürfen. Wird ein Zug gewählt, von dem nur der Gegner profitiert, sollten zukünftige Simulationen durch den neuen Knoten einen geringen Mittelwert ergeben. Allerdings ist die Neugierigkeitsheuristik aus der UCT-Formel (3.1) schlimmsten Falls durch Δ_v groß genug, um häufig simuliert zu werden. Dies kostet Zeit für wichtigere Züge und kann sogar dazu führen, dass sich der bedrohliche Zug durchsetzt.

Die First-Play-Urgency kann mit der nachfolgenden Formel berechnet werden:

$$\mathrm{FPU}(v_k) := \begin{cases} c & \text{für } k \in \{0,1\} \\ v_{k-2i}.\bar{x} + (c - 1.0) & \text{wenn } v_{k-2i}.n \geq t \\ -\infty & \text{sonst} \end{cases} \tag{3.2}$$

v_k beschreibt einen Knoten auf der k-ten Stufe. v_{k-2i} sei der nächste Knoten für den selben Zug auf dem Weg zur Wurzel. $c - 1.0$ ist eine definierte, konstante Neugierde für nicht besuchte Knoten. Da es für die ersten beiden Ebenen keinen Vorgänger geben kann, wird $\bar{x}$ wie für die konstante FPU maximal gewählt. t ist die beschriebene Schranke, um eine minimale Genauigkeit für die Schätzwerte sicher zu stellen. Gibt es keine zulässige Referenz, nimmt die Urgency den kleinsten Wert an. Von Knoten mit Kindern wird

immer jenes mit dem höchsten UCT-Wert gewählt. Für ein Blatt muss ein neuer, zufälliger Zug gewählt werden.

3.3.3 Simulationswerte vorziehen

UCT simuliert Zugfolgen, so dass der Verlust durch das Simulieren eines nicht optimalen Zugs klein wird. Dafür sorgt das UCB1-Verfahren. UCB1 berücksichtigt aber nur die Knoten, die bereits simuliert wurden. Für neue, nicht erzeugte Knoten muss der Verlust durch die First-Play-Urgencies abgeschätzt werden. Nachteil der letzten FPU ist, dass alte Werte benutzt werden. Eine Alternative zu einem guten Zug muss daher bereits zuvor dem guten Zug relativ ähnlich gewesen sein. Die bisher besten Züge werden jedoch in der Regel gespielt und können meistens nicht für die FPU herangezogen werden. Der Verlust gegenüber dem tatsächlich besten Zug lässt sich daher nur schwer abschätzen.

Anfängern wird für Go häufig geraten, nach dem besten Zug des Gegners zu suchen. Auf diese Weise fängt der Spieler an zu planen. Nicht nur der eigene Zug ist entscheidend. Auch die daraus folgenden Situationen für den Gegner fallen ins Gewicht. Auf einen schwachen Zug eines Spielers, hat sein Gegner die Möglichkeit einen stärkeren Zug zu spielen. Es stellt sich heraus, dass der starke Zug des Gegners auch häufig ein starker Zug für den Spieler selbst gewesen wäre. Diese Beobachtung könnte eine gute Abschätzung für den Verlust vom momentan besten Zug gegenüber einem noch nicht erforschten Knoten sein.

Für diese First-Play-Urgency werden Folgezüge als Alternativen herangezogen. Im Gegensatz zu Abbildung 3.1 wird nun der beste Nachfolger betrachtet (Abbildung 3.2). Die Entscheidung für den Referenzzug ähnelt der Killer Heuristik. Die Killer Heuristik ist ein Spezialfall der History Heuristik. Es wird nur nach einem Zug - dem Killermove - gesucht, der sich in der Anwendung als FPU dadurch auszeichnet, dass er in einer späteren Stellung vorteilhaft sein wird. Tiefere Knoten im Baum wurden zwar seltener simuliert. Andererseits befinden sie sich auch näher an einer Endstellung.

Es können nun auch gefahrloser gegnerische Züge als Referenz herangezogen werden. Sobald ein gegnerischer Zug gefährlich wird, bekommen die vorherigen Züge des Spielers wahrscheinlich eine schlechte Wertung. Ein gefährlicher Zug geht dann nicht in die FPU ein, weil nur der beste Pfad bezüglich des Mittelwerts verfolgt wird.

Die Berechnung der First-Play-Urgency erfolgt ähnlich zur Definition 3.2:

$$\mathrm{FPU}(v_k) := \begin{cases} v_{k+i}.\bar{x} + c - 1.0 & \text{wenn } v_{k+i}.n \geq t \\ -\infty & \text{sonst} \end{cases} \tag{3.3}$$

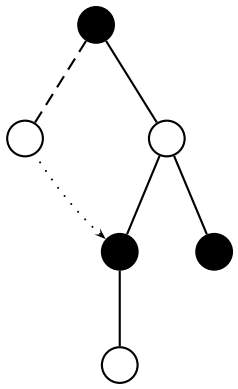

Abbildung 3.2: Der Algorithmus prüft, ob ein neuer, weißer Knoten untersucht werden soll. Der neue Knoten ist ein nicht expandierter Zug vom bisher besten Pfad. Der beste Pfad besteht in diesem Beispiel aus vier Knoten. Der Simulationswert wird als FPU verwendet.

Wie zuvor beschreibt v_k einen Knoten auf der k-ten Stufe. v_{k+i} sei der nächste Knoten für den selben Zug auf dem besten Weg zu einem Blatt. Der beste Weg wird über den Mittelwert bestimmt. Der Mittelwert eines gegnerischen Zugs wird nicht negiert. Ein guter Zug des Gegners soll hier auch ein guter Zug des Spielers sein. Schlechte Züge, sind meistens für beide Spieler schlecht. Hierzu zählen zum Beispiel die Züge in die Ecken am Anfang eines Spiels. c kann unterschiedlich zu Definition 3.2 gewählt werden. t ist die Schranke, um eine minimale Genauigkeit für die Schätzwerte sicher zu stellen. Um die Definition 3.3 zu vereinfachen, wird t für unerforschte Knoten auf 0 gesetzt. Für Blätter nimmt die Urgency daher den kleinsten Wert an. Es wird ein neuer, zufälliger Zug gewählt.

Die Definition 3.3 berechnet für einen neuen Knoten den erwarteten Wert. Die First-Play-Urgency lässt sich deutlich effizienter implementieren, wenn der neue Knoten aus dem maximalen Wert des besten Pfads bestimmt wird. Wenn ein neuer Zug mit maximalem Wert nicht expandiert wird, gilt dies auch für alle anderen Züge mit kleineren Werten. Wenn ein Knoten gefunden wird, dessen Zahl an Simulationen kleiner als t ist, kann die Suche abgebrochen werden. Weit entfernte Züge werden wenig auf den betroffenen Spielstand übertragbar sein. Daher kann die Suche eventuell auch früher abgebrochen werden.

Kapitel 4

Evaluation

In den vorangegangenen Kapiteln wurden verschiedene First-Play-Urgencies vorgestellt. Ziel der FPUs ist, das Spielverhalten zu verbessern, ohne tiefer gehendes Expertenwissen einfließen zu lassen. Dieser Abschnitt vergleicht die vorgestellten Verfahren. Die Evaluation wird auf einem Datenset aus 20 Go Spielen durchgeführt. Die 20 Spiele sind eine Auswahl aus den Spielen von (van der Steen, 2002). Jede Partie wurde auf einem Brett mit 9×9 Kreuzungen gespielt und enthält über 50 Züge. Die gesamte Datenmenge besteht aus 1151 Zügen. Der Rang der Spieler liegt zwischen 7 und 9 Dan. Kyu sind die Schüler- und Dan die Meisterränge. Anfänger beginnen bei etwa 20 bis 30 Kyu. Der Kyu-Rang nimmt ab, wenn sich der Spieler verbessert. Ein guter Vereinsspieler erreicht 1 Kyu. Der nächstbeste Rang ist der erste Meisterrang - Der erste Dan. Der Dan-Rang steigt, wenn sich der Spieler verbessert.

Jede Variante der UCT-Suche wird im Folgenden als eigener Algorithmus betrachtet. Für jeden Algorithmus wird das gleiche Experiment durchgeführt. Der Algorithmus versucht den nächsten Zug des menschlichen Spielers zu erraten. Dieser Zug wird nun als Meisterzug bezeichnet. Der Algorithmus spielt abwechselnd für beide Spieler. Die möglichen Aktionen werden für jeden Zug nach der Anzahl ihrer Simulationen absteigend sortiert. Gesucht ist die Position der Meisterzüge in den sortierten Listen. Bei gleicher Zahl an Simulationen wird dem Meisterzug immer der schlechtere Rang zugeordnet. Dies ist eine pessimistische Bewertung und soll den Algorithmus dafür bestrafen, dass er die Züge nicht voneinander abgegrenzt hat. Ein naiver Algorithmus, der alle Züge gleich oft simuliert, würde daher immer schlecht bewertet werden. Wurde der Meisterzug nie simuliert, bekommt er den schlechtesten Rang. Sei M_k die Menge der legalen Züge nach k gespielten Zügen, so sei der schlechteste Rang mit $|M_k|+1$ definiert. Für den zufälligen Spieler gilt also, dass der Meisterzug entweder zufällig erraten wird und somit in der Liste an erster

Stelle steht oder verfehlt wird und mit $|M_k| + 1$ bewertet wird.

Der Meisterzug führt zum nächsten Zustand. Wieder muss der Algorithmus den nächsten Zug erraten, bevor dieser gespielt wird.

Die Signifikanztests auf den Experimenten sollen hier nur knapp erklärt werden. Zuerst wird der Friedman-Test durchgeführt. Dieser sortiert die Algorithmen nach der Güte ihrer Schätzung. Wenn sich die Algorithmen nicht unterscheiden, sollte deren Ordnung zufällig sein. Durch die Zuweisung von Rängen wird die Verteilung, die den Algorithmen zugrunde liegt, irrelevant. Geprüft wird, ob die Ordnung der Algorithmen zufällig ist. Die Positionen sind annähernd χ^2 verteilt. Es lässt sich der F-Wert berechnen, der für n (= 1151 Züge) > 10 eine genauere Schätzung für das Signifikanzniveau liefert (Demšar, 2006). Der Friedman-Test ist eine Verallgemeinerung des Sign-Tests. Es können mehrere Algorithmen verglichen werden. Die Nullhypothese nimmt an, dass es keinen signifikanten Unterschied zwischen den getesteten Algorithmen gibt. Wird die Hypothese verworfen, gibt es mindestens zwei Familien von Algorithmen, deren Ordnung nicht zufälliger Natur ist.

Die Anzahl der Familien und deren Elemente können durch die kritischen Distanzen zwischen den Algorithmen bestimmt werden. Die kritischen Distanzen sind abhängig vom Signifikanzniveau, der Größe der Stichprobe und der Anzahl der Algorithmen. Der Wert einer Distanz kann mit zwei Tabellen analysiert werden. Die Distanzen werden mit dem Nemenyi-Test geprüft, wenn alle Algorithmen untereinander verglichen werden sollen. Wenn nur ein Unterschied zu einem anderen Verfahren gezeigt werden soll, verwendet man den Bonferroni-Dunn-Test. Dieser ist genauer und daher vorzuziehen, wenn ein Referenzalgorithmus herangezogen werden kann (Demšar, 2006). Für beide Tests gibt es eine eigene Tabelle. Ist eine kritische Distanz kleiner als der entsprechende Wert aus der Tabelle, gehören die Algorithmen wahrscheinlich zur selben Familie. Andernfalls gibt es wahrscheinlich einen signifikanten Unterschied. Es kann passieren, dass die Stichprobe keine klare Aussage ermöglicht.

Um die Qualität der Schätzungen zu beurteilen, werden zwei Darstellungen verwendet. Die erste Darstellung betrachtet die Trefferrate. Dazu wird gezählt wie oft der Algorithmus den Meisterzug korrekt erraten hat. Dann wird das Verhältnis zu der Gesamtzahl an Versuchen (1151) bestimmt. Die zweite Darstellung betrachtet die Qualität der Schätzung. Hier wird der Bereich gesucht, in dem der Meisterzug durchschnittlich zu finden ist. Die Qualität wird berechnet durch

$$\text{quality}(algo) := \frac{1}{n} \cdot \sum_{k=1}^{n} \frac{\#\text{free}(k) - \text{rank}_{algo}(k)}{\#\text{free}(k)}.$$

#free(k) ist die Zahl an freien Kreuzungen des k-ten Beispiels. $\text{rank}_{algo}(k)$ ist die Position in der Liste, die durch den Algorithmus erstellt wird. Taucht das Beispiel nicht in der Liste auf, wird ist die Qualität der Schätzung für dieses Beispiel mit Null definiert.

4.1 Der Einfluss der Simulation

UCT profitiert vom Gesetz der großen Zahlen. Es ist bewiesen, dass UCB1 gegen die optimale Lösung konvergiert, wenn genug Zeit investiert wird. Demnach wächst die Stärke des Computer-Spielers mit der Anzahl der Iterationen. In Abbildung 4.1 wird die Stärke des Computer-Spielers gemessen.

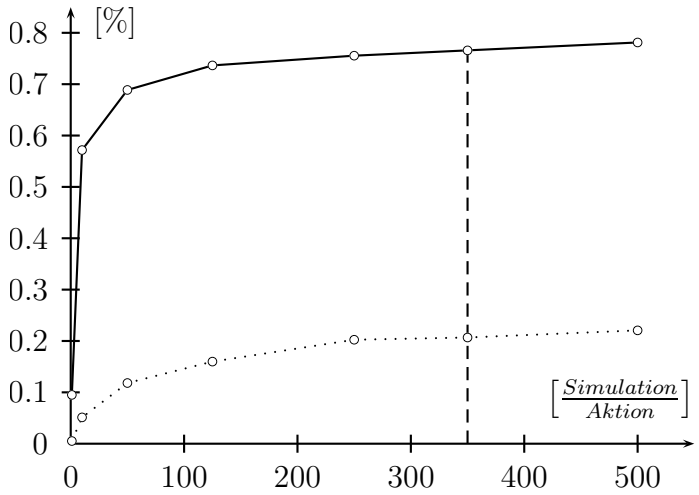

Abbildung 4.1: Mit steigender Zahl an Simulationen wird ein größerer Anteil an Meisterzügen erraten. Die Stärke eines UCT-Spielers steigt stetig mit der Zeit. Die durchgezogene Linie zeigt die Qualität der Schätzungen und die gepunktete Linie die Trefferrate. Für weitere Experimente werden 350 Simulationen pro Aktion durchgeführt.

Für die Messung wurde UCT mit der konstanten First-Play-Urgency parametrisiert. Die Konstante der FPU wird groß gewählt (10000), so dass stets alle Kinder eines Knotens erzeugt werden. Im folgenden Abschnitt wird der Neugierigkeitssummand der UCT-Formel (3.1) skaliert. Noch wird allerdings keine Skalierung verwendet. Untersucht wird, wie sich die Spielstärke in Bezug zur Zahl an Simulationen ändert. Für Stellungen, in denen viele Aktionen möglich sind, sollen mehr Simulationen durchgeführt werden. Dies wird erreicht, indem lediglich die Zahl an Simulationen pro Kreuzung festgelegt wird. Je mehr freie Kreuzungen in einer Stellung existieren, desto mehr

Simulationen werden in diese investiert.

Die Stärke steigt stetig mit der Zahl an Simulationen. Im Folgenden werden 350 Simulationen pro Aktion verwendet. Dies bedeutet allerdings nicht, dass jede Aktion 350 mal simuliert wird. UCT versucht weiterhin gute Züge häufiger zu simulieren als Schlechte. Der gewählte Wert bietet einen ausreichenden Kompromiss zwischen Zeit und Genauigkeit. Im Durchschnitt werden für jede Schätzung eines Meisterzugs etwa 20000 Simulationen investiert.

4.2 Der Einfluss von UCB1

UCB1 ist für das Balancieren des Exploration-Exploitation-Problems verantwortlich. Durch Skalierung des Neugierigkeitssummanden, wird gesteuert, ob der Algorithmus eher zum Erkunden neuer Varianten oder zum Ausschöpfen ertragreicher Züge neigt. Das Ausschöpfen eines Zugs äußert sich darin, dass auf dessen Grundlage eine Strategie konstruiert wird.

Das Experiment analysiert die Extrema, sowie einige Werte im optimalen Bereich Abbildung 4.2. Die Extremwerte sollen zeigen, wie sich UCT verhält, wenn entweder der Gier nach guten oder neuen Werten gefolgt wird. Die Trefferrate nimmt für eine Verstärkung der Neugierde bei etwa 1.2 ein Maximum an. Wählt der Algorithmus immer den Zug mit dem besten Mittelwert, ist das Ergebnis besser als die Züge gleichmäßig zu untersuchen. Diese Beobachtung ist nicht sehr überraschend, weil der beste Zug anhand der Simulationsanzahl bestimmt wird. Das gleichmäßige Erkunden der Züge wird nur geringfügig vom tatsächlichen Ertrag des besten Zugs beeinflusst. Dadurch ist die Auswahl des zu spielenden Zugs letztendlich nahezu zufällig.

Durch den Signifikanztest über die Ordnung der Algorithmen, lassen sich vier Familien feststellen. Der Wert 1.2 hat eine höhere Trefferrate und eine niedrigere Qualität als 1.5. Weil die Trefferrate strenger ist, wird im Folgenden der Wert 1.2 verwendet. Man kann in Abbildung 4.2 ebenfalls erkennen, dass die Familien in der Qualitätsmessung schwieriger zu sehen sind. Der Wert 1.2 könnte in seiner Familie durch Zufall die maximale Trefferrate angenommen haben. Die Testmenge reicht nicht aus, den Wert signifikant abzugrenzen.

4.3 Variation der konstanten FPU

Das Ziel der konstanten First-Play-Urgency ist eine längere Strategie für den besten Zug zu finden. Je kleiner die Konstante ist, desto häufiger wird ein bereits untersuchter Zug ein weiteres Mal gewählt. Ist die Konstante höchstens

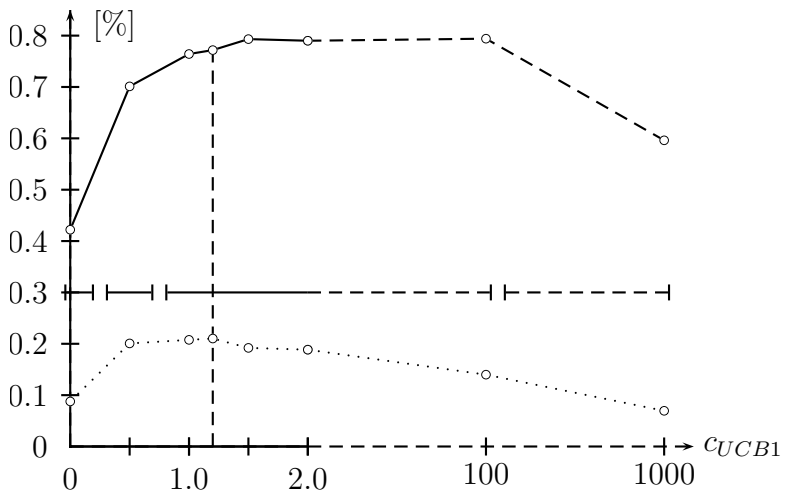

Abbildung 4.2: Die durchgezogene Kurve zeigt die Qualität und die Gepunktete die Trefferrate. Die Genauigkeit wächst, bis der Skalierungsfaktor etwa den Wert 1.2 erreicht. Die Intervalle geben an, welche Werte statistisch gleich sind. Die Hypothese, dass es keinen Unterschied zwischen zwei Gruppen gibt, kann mit einer Wahrscheinlichkeit von mindestens 95% verworfen werden. Während die Kurve den Anteil der korrekten Vorhersagen angibt, wurden die Intervalle mit dem Friedman-Test über alle Ränge bestimmt.

Null, wird eine zufällige Strategie verfolgt. Der Algorithmus expandiert genau einen Pfad. Für eine Konstante, die größer als jeder Simulationswert ist, wird niemals in einen erkundeten Zug abgestiegen, bevor nicht alle Alternativen einmal simuliert wurden. Abbildung 4.3 zeigt die Genauigkeit für die verschiedenen Konstanten.

Das Maximum bei 1.4 lässt sich nicht signifikant von großen Konstanten (hier 10000) abgrenzen. Dennoch kann man vermuten, dass eine kleinere Konstante sich im Spielverhalten besser verhält. Beim Erraten vorgegebener Zugfolgen erkennt man jedoch keine Verbesserung. Abbildung 4.4 zeigt, wie sich die Bäume für die Konstanten 1.4 und 10000 nach 56700 Iterationen (700 Iterationen pro Kreuzung auf dem leeren Brett) unterscheiden.

Je kleiner die Konstante wird, desto tiefer sucht UCT nach einer Strategie. Diese ist jedoch für zu kleine Werte immer mehr dem Zufall unterworfen. Da im Optimalfall mehr Zeit für die beste Zugfolge investiert wird, ist das Verfahren auch empfindlicher gegenüber Abweichungen zur gefundenen Lösungen. Irrt sich der Spieler, kann er durch seinen Gegner überrascht werden.

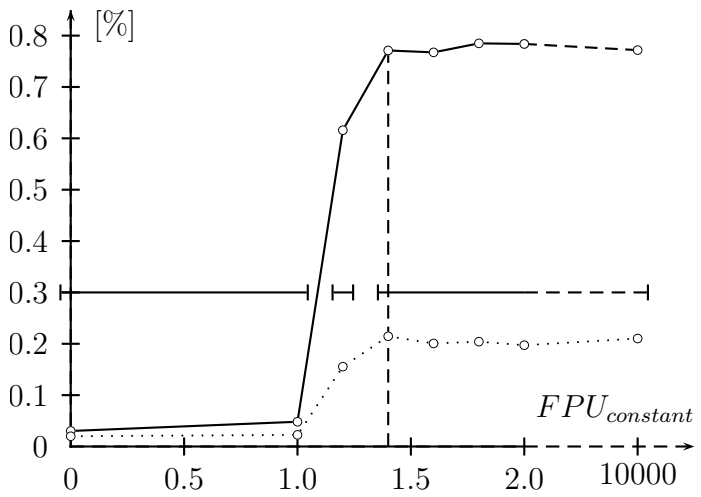

Abbildung 4.3: Die Qualität (durchgezogen) und Trefferrate (gepunktet) wächst mit der Größe der Konstanten. Ab etwa 1.4 gibt es keine signifikante Verbesserung mehr. Die Intervalle über der Kurve kennzeichnen, welche Werte durch die kritische Distanz zu einer Gruppe zusammengefasst werden können. Die Hypothese, dass es keinen Unterschied zwischen den Gruppen gibt, kann jeweils mit 99%-iger Sicherheit verworfen werden.

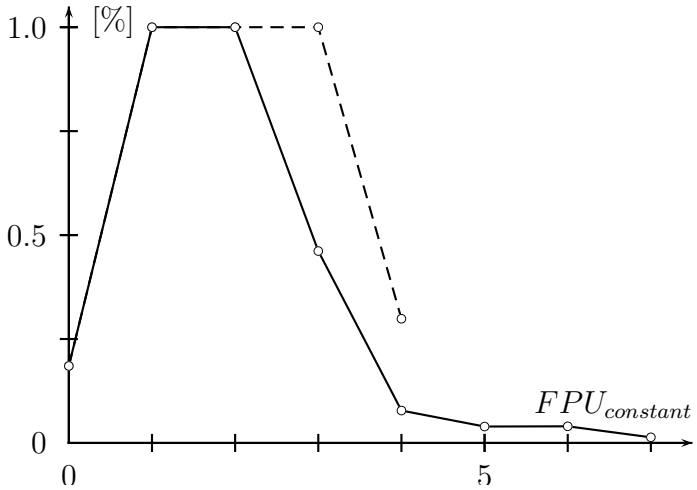

Abbildung 4.4: Die Kurven illustrieren wie viel Prozent der möglichen Züge für die Konstanten 1.4 und 10000 expandiert werden. Dabei wird der beste Pfad im entsprechenden Baum betrachtet. Die durchgezogene Linie bezieht sich auf den Wert 1.4 und die gestrichelte Linie auf den Wert 10000. Vom leeren Brett (Ebene 0) wird aus Symmetriegründen nur ein Achtel der Züge untersucht.

4.4 Variation der vererbenden FPU

Bei der vererbenden FPU gibt es zwei Parameter, die optimiert werden müssen. Wie bei den vorherigen Verfahren wird eine Konstante gesucht, die entscheidet, ob ein bereits simulierter oder ein neuer Knoten untersucht werden soll. Dazu kommt die Zahl an Simulationen, die durchgeführt werden müssen, damit ein Knoten als Referenz herangezogen wird. Der Grenzwert gibt an, ab wann der Mittelwert nicht mehr als instabil gesehen wird. Die Evaluation wird dadurch erschwert. Die Konstante ist vom Grenzwert abhängig, weil durch den Grenzwert die Neugierigkeitssummanden der bereits simulierten Knoten variieren.

Der Grenzwert wird daher zuvor auf 20 Iterationen festgelegt. Abbildung 4.5 zeigt die Veränderung der Genauigkeit für die festgelegte Grenze und verschiedene Konstanten. In der Abbildung erkennt man, dass die Qualitätsmessung ein glatteres Maß ist. Man sieht ebenfalls, dass sich die Werte bis 2.0 ähnlicher sind, als die Trefferrate suggeriert.

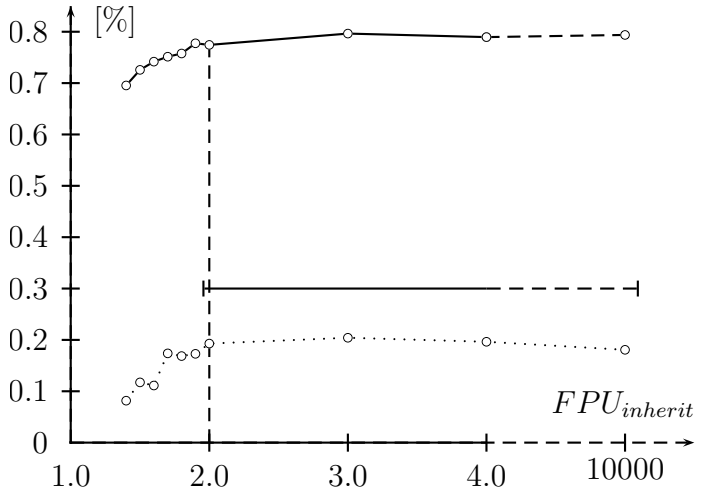

Abbildung 4.5: Die Kurve zeigt die Treffergenauigkeit für die vererbende FPU. Der Grenzwert ist mit 20 Iterationen definiert. Ab der Konstanten 2.0 bildet sich eine Gruppe, die statistisch gleich ist. Davor geben die kritischen Distanzen keinen Aufschluss über die tatsächliche Gruppierung, weil sich die Intervalle überschneiden. Die Werte 1.4, 1.5, 1.7 und 2.0 unterscheiden sich jedoch signifikant auf dem 95%-Niveau.

Während der Arbeit ist ein Hilfswerkzeug entstanden, mit dem es möglich ist die Ebenen des simulierten Baums zu betrachten. Es verdeutlicht, welche Züge auf der gewählten Ebene expandiert wurden und liest beispielsweise aus wie viele Simulationen diese erfahren haben (Abbildung 4.6). Beim Ver-

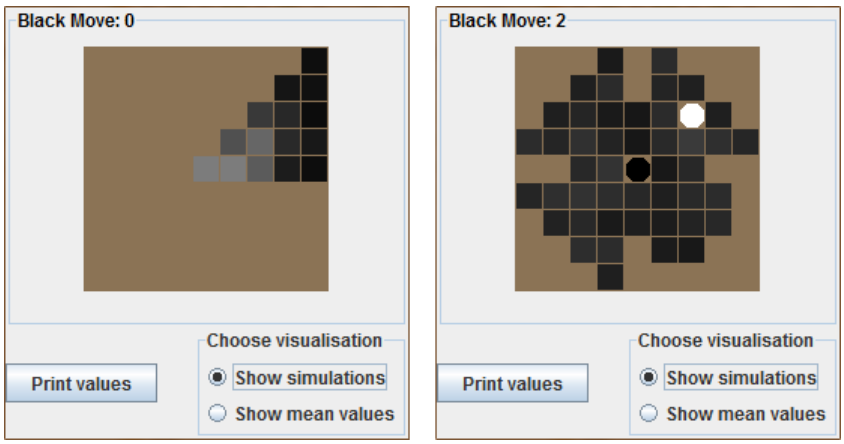

Abbildung 4.6: Die Abbildung zeigt das Tool, mit dem der simulierte Spielbaum untersucht werden kann. Auf der linken Seite wird gezeigt, welche Knoten UCT für die Wurzel untersucht hat. Je heller die Fläche ist, desto häufiger wurde eine Variante simuliert. Auf der rechten Seite erkennt man, welche Züge ausprobiert wurden, nachdem Schwarz in die Mitte und Weiß auf $(7, 7)$ gespielt hat. Die Simulation ist auf das Zentrum fokussiert, weil die FPU auf die Simulationswerte der Wurzel zurückgegriffen hat. Züge in die Ecken werden vermieden.

erben von Simulationswerten fällt auf, dass vor allem in den frühen Ebenen verhindert wird in die Ecken zu setzen. Die Suche fokussiert sich also automatisch auf das mittlere Spielfeld. Je größer die Konstante gewählt wird, desto mehr werden auch wieder die Randbereiche betrachtet. Auch wenn sich das Verfahren in dieser Evaluation nicht durchsetzt, lässt sich eine deutlich gezieltere Auswahl der nächsten Züge erkennen. Ein weiterer Vorteil ist, dass die Symmetrie der ersten Ebene für die Folgenden verwendet werden kann.

4.5 Variation der vorziehenden FPU

Die FPU bei der zukünftige Werte für die Abschätzung herangezogen werden, soll ausnutzen, dass der Gegner eventuell von einem schwachen Zug profitiert. Durch das Erkunden dieses Zugs von einer früheren Ebene, kann untersucht werden, ob die Variante sich bereits früher als vorteilhaft erweist. Wie bei der vererbenden FPU müssen zwei Werte variiert werden, von denen der Grenzwert für die Stabilität der Mittelwerte wieder auf 20 festgelegt wird. Dieser Wert wurde nur mit Hilfe des Tools aus Abbildung 4.6 abgeschätzt. Wie sich das Spielverhalten für die verschiedenen Konstanten verhält, zeigt Abbildung 4.7

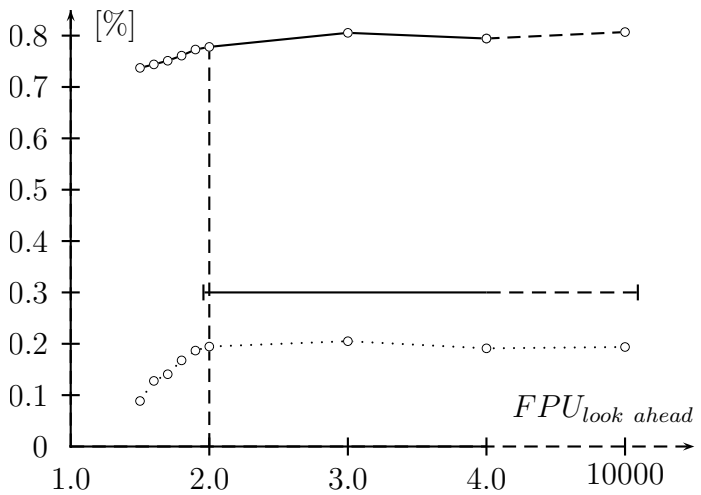

Abbildung 4.7: Die Kurve zeigt die Qualität (durchgezogen) und die Trefferrate (gepunktet) für die vorziehende FPU. Der Grenzwert ist mit 20 Iterationen definiert. Ab der Konstanten 2.0 bildet sich eine Gruppe, die statistisch gleich ist. Davor geben die kritischen Distanzen keinen Aufschluss über die tatsächliche Gruppierung, weil sich die Intervalle überschneiden.

4.6 Schlussfolgerung

Die Ergebnisse lassen keinen klaren Gewinner hervorgehen. Es lassen sich dennoch einige Aussagen über den Algorithmus treffen.

UCT ist auf allgemeine Baumsuchprobleme anwendbar. Diese und verwandte Arbeiten wie unter Anderen (Brügmann, 1993), (Coulom, 2006) und (Gelly & Wang, 2006) haben gezeigt, dass die Monte-Carlo-Suche und UCT auch auf Problemen mit vollständiger Information erhebliche Vorteile bringt. Die Evaluation zeigt deutlich, dass die Qualität der Berechnungen automatisch mit der Stärke der ausführenden Maschinen wächst. Bereits der unveränderte Algorithmus erreicht eine hohe Genauigkeit. Die Meisterzüge sind durchschnittlich im oberen Viertel zu finden.

Einen deutlichen Einfluss durch die FPU lässt sich nicht beobachten. Die Tabelle 4.1 stellt die gewählten Kandidaten der verschiedenen Verfahren gegenüber. Die Qualität der Schätzungen unterscheiden sich erst in der dritten Dezimalstelle. Weiterhin kann man erkennen, dass die Ergebnisse der Qualität den Ergebnissen der Trefferrate widersprechen. Auch dies ist ein Indiz, dass die Verfahren unter dem verwendeten Test als gleichwertig angesehen werden müssen.

Die drei letzten Verfahren expandieren selten alle Knoten. Daher ist es möglich, dass der Meisterzug nicht gefunden wird. In diesem Fall bekommen

Verfahren	**Qualität**	**Trefferrate**
Konstant (∞ / `M`)	0.7718	0.2103
Konstant (1.4)	0.7712	0.2146
Vererbt (2.0)	0.7744	0.1929
Voraus geschaut (2.0)	0.7781	0.1946

Tabelle 4.1: Die Tabelle zeigt die Genauigkeitswerte der gewählten Parametrierungen. Die Werte einer Spalte unterscheiden sich nicht signifikant. Durch den Friedman-Test ergibt sich ein F-Wert von 0.81. Der Wert für das 95%-Signifikanzniveau 8.53 wird deutlich unterschritten. Auch die kritischen Distanzen können keine Gruppen unterscheiden.

die Verfahren einen schlechten Rang. Damit man besser den Einfluss der verschiedenen FPUs erkennen kann, wurde das Experiment ein wenig verändert. Die Veränderung besteht darin, dass der Meisterzug künstlich als erster Zug expandiert wird. Der Rang des Meisterzugs bestimmt sich dann lediglich anhand der Zahl an Simulationen. In Tabelle 4.2 sind die Ergebnisse des veränderten Experiments aufgeführt. Da sich keine signifikanten Unterschie-

Verfahren	**Qualität**	**Trefferrate**
Konstant (∞ / `M`)	0.7915	0.2103
Konstant (1.4)	0.7698	0.2007
Vererbt (2.0)	0.7719	0.1920
Voraus geschaut (2.0)	0.7807	0.1833

Tabelle 4.2: Die Tabelle zeigt die Genauigkeitswerte der gewählten Parametrierungen. In dem entsprechenden Experiment, wurde der Meisterzug immer als erster gefunden. Die Werte einer Spalte unterscheiden sich nicht signifikant. Sie sind zum Teil sogar niedriger als im ursprünglichen Experiment. Durch den Friedman-Test ergibt sich nun ein F-Wert von 2.32. Der Wert für das 95%-Signifikanzniveau 8.53 wird immer noch deutlich unterschritten. Auch die kritischen Distanzen können keine Gruppen unterscheiden.

de erkennen lassen, ist es lediglich möglich den Unterschied im Suchverhalten zu analysieren.

Wie bereits zuvor in Abbildung 4.4 zu sehen war, erreicht man mit einer

kleineren Konstante eine längere Strategie. Im optimalen Fall kann man diesen Vorteil nutzen, um schneller voraus zu planen. Der größte Gewinn wird jedoch dadurch erreicht, dass der Speicherbedarf sinnvoller genutzt wird. Je schneller gute Züge gefunden werden, desto weniger neigt die Suche dazu neue Alternativen zu erzeugen. Dadurch ist der Algorithmus nicht gezwungen alle Eck- und Randbereiche zu untersuchen, bis diese potentiellen Nutzen bringen. Dies liegt aber lediglich daran, dass die Anzahl der Kreuzungen am Rand niedriger ist als in der Mitte. Somit ist die Wahrscheinlichkeit, dass zu Beginn ein Zug am Rand gewählt wird nur

$$Pr(\text{Untersuche Zug am Rand}) = \frac{4(n-1)}{n^2} \overset{(9\times 9)}{\approx} 0.395.$$

Die Wahrscheinlichkeit nimmt linear mit der Brettgröße ab.

Die Wahrscheinlichkeit wird durch die vererbende FPU gezielter gesteuert. Ein Nachteil ist, dass von alten Simulationswerten auf Neue geschlossen werden muss. Allerdings zeigt sich in der Form des Spielbaums, dass diese Variante sehr gut die Symmetrie der ersten Ebenen ausnutzen kann. Auf dem leeren Brett muss nur etwa ein Achtel aller Züge untersucht werden. Durch Rotation und Spiegelungen der Züge kann man die Simulationswerte dann auf die äquivalenten Kreuzungen übertragen. Sogar diese Symmetrieeigenschaft wird in den späteren Ebenen automatisch weiterverwendet (Abbildung 4.6). Die Vererbung von Simulationswerten würde sich eventuell deutlicher bemerkbar machen, wenn sie noch einen Einfluss auf wenig simulierte Züge hätte. Eine nahe liegende Möglichkeit wäre, Züge initial mehrmals hintereinander zu simulieren. Die vererbten Werte könnten dann dazu benutzt werden, die Zahl an initialen Simulationen zu bestimmen. Diese Möglichkeit wurde in dieser Arbeit noch nicht untersucht.

Eine weitere Variation der FPU mit einer kleinen Konstante stellt die vorziehende FPU dar. Interessant ist, dass auch hier in den ersten Ebenen symmetrische Strukturen entstehen. Allerdings kann dabei nicht das Wissen aus der ersten Ebene einfließen. Die FPU kann erst in den späteren Ebenen wirklich profitieren. Bei Stellungen in denen ein Fehler große Auswirkungen hat, kann schnell ein schlechter Zug durch einen Guten ausgetauscht werden. Im optimalen Fall ist der Zug, den ein Spieler gegen den Anderen benutzt, der Zug der zu einem früheren Zeitpunkt benötigt wird.

Es gibt jedoch einige Aspekte, die gegen den Einsatz der vorziehenden FPU sprechen. Zum Einen ist der Einsatz stärker an das Spiel Go gebunden als die vorherigen FPUs. Dies hängt damit zusammen, dass gute Züge für den einen Spieler nicht zwangsweise auf den Anderen übertragbar sind. Auf Problemstellungen in denen konkurrierende Agenten unterschiedliche Aufgabenbereiche haben, ist diese FPU gar nicht anwendbar. Dazu wird der

Aufwand beim Absteigen zum vorläufig optimalen Blatt zu $O(n^2)$. Dies liegt daran, weil der nächste Zug nach dem UCT-Wert und der beste Zug anhand seines Mittelwerts ausgesucht wird. Im schlimmsten Fall muss für jeden Knoten eine neue Referenz gesucht werden. Diese wiederum ist schlimmsten Falls ein Blatt.

Am erfolgversprechendsten scheint daher die vererbende FPU zu sein. Sie ist in der Lage bekannte Strukturen automatisch auf folgende Situationen zu übertragen. Dazu ist es möglich sich auf die speziellen Aktionen des entsprechenden Agenten zu beschränken. Die vererbende FPU kann einfach in weitere Bereiche der Baumsuche integriert werden. Es lässt sich eine geplante Suche erkennen. Hierdurch entsteht der Eindruck, dass das Verfahren aus der Geschichte einer Situation eigenes Expertenwissen generiert.

Kapitel 5

Zusammenfassung

Im Mittelpunkt der Arbeit steht die Repräsentation des Spielzustands, die UCT-Suche und Heuristiken mit denen die Suche verbessert werden soll.

Der Spielzustand ist in wichtige Aspekte für die künstliche Intelligenz zerlegt worden. Zu jedem Aspekt wurden die wesentlichen Schritte für eine günstige Implementierung vorgestellt. Die Wahl der Implementierung wurde erläutert und zum Teil Alternativen gegenübergestellt.

Für die Suche nach der besten Zugfolge wurde in dieser Arbeit eine UCT-Suche verwendet. Die UCT-Suche baut auf dem Monte-Carlo-Verfahren auf. Das Monte-Carlo-Verfahren ist in die UCT-Suche übergeleitet und die Teilschritte der Suche auf einer abstrakten Ebene erklärt worden.

Durch Heuristiken wurde versucht den Spielbaum effizient aufzubauen. Ziel war es, die Suche in die Breite zu einer Suche in die Tiefe zu bewegen. Dazu wurden vier Heuristiken vorgestellt: Zwei Heuristiken die mit verschiedenen, konstanten Werten arbeiten, eine History Heuristik und eine Killer-Heuristik. In dieser Arbeit hat sich herausgestellt, dass die vorgestellten Heuristiken die Suche nicht signifikant beeinflussen, wenn man die Spielstärke betrachtet. Allerdings konnten deutliche Unterschiede im Aufbau des Spielbaums beobachtet werden.

Anhang A

Don'ts

Die Arbeit hat eine mögliche Implementierung eines Computer-Go-Spielers vorgestellt. Während der Implementierungszeit sind einige Fehler entstanden, die sich stark auf das Spielverhalten auswirken und dennoch nicht immer leicht zu erkennen sind. Mit den besten Wünschen für alle folgenden Arbeiten, sei hier vor zwei Implementierungsfallen gewarnt.

A.1 Vertauschen der Simulationsergebnisse

Nachdem ein Blattknoten simuliert wurde, wird das Ergebnis über jeden Knoten des ausgewählten Pfads zur Wurzel hoch gereicht. Es ist wichtig zu beachten welcher Knoten, mit welchem Ergebnis aktualisiert wird.

Abbildung A.1 zeigt, wie ein Simulationsergebnis in die Wurzel propagiert wird. Am Ende der Simulation gewinnt der schwarze Spieler. Nur die Züge

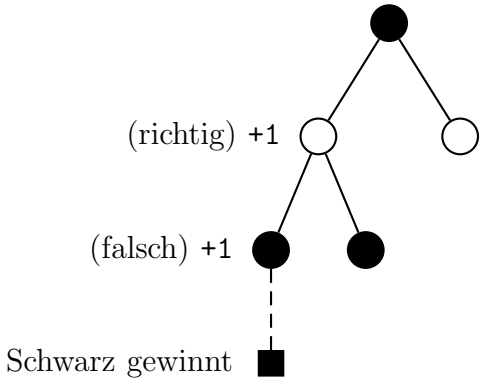

Abbildung A.1: Das simulierte Spiel ist als Viereck dargestellt. Schwarz gewinnt. Die Züge des schwarzen Spielers sollen durch die Addition einer `1` belohnt werden.

von Schwarz dürfen eine Belohnung bekommen. Es werden jedoch nicht die schwarzen Knoten belohnt. Der schwarze Spieler wählt mit einem Zug einen weißen Knoten aus. Die Ergebnisse für Schwarz müssen daher in einem weißen Knoten oder der Kante zum weißen Knoten gespeichert werden. In der Implementierung zu dieser Arbeit wurden keine Kanten modelliert. Werden die Knoten mit den falschen Ergebnissen aktualisiert, findet der Algorithmus näherungsweise den schlechtesten Zug. Der Fehler lässt sich leicht erkennen, kann aber auch ebenso leicht entstehen.

A.2 Den besten Zug spielen

Nach der Simulationszeit spielt der Algorithmus den besten Zug, den er gefunden hat. Der beste Zug wird durch die Zahl an Simulationen bestimmt. Während der Simulationszeit werden die Züge allerdings nach dem höchsten UCT-Wert ausgewählt. Dadurch haben auch Werte mit niedrigem Mittelwert eine Chance simuliert zu werden. Ein hoher UCT-Wert gibt an, dass der entsprechende Zug als nächstes simuliert wird.

Während der Implementierung des Algorithmus wird häufig der UCT-Wert als Auswahlkriterium verwendet. Es kann aus Gewohnheit passieren, dass auch bei der Auswahl des zu spielenden Zugs anhand des UCT-Werts anstatt der Zahl an Simulationen entschieden wird. Dieser Fehler ist zu Beginn unauffällig. Gute Züge werden asymptotisch exponentiell häufig simuliert. Unter vielen Zügen passiert es daher selten, dass ein schlechter Zug einen hohen UCT-Wert bekommt. Gegen Ende eines Spiels wechselt das Spielverhalten. Je weniger Züge zur Auswahl stehen, desto stärker deuten sich zufällige Entscheidungen an. So ist es möglich, dass der gewinnende Spieler beim Ausspielen der letzten Punkte noch wichtige Gruppen opfert und dadurch verliert.

Literaturverzeichnis

Auer, P., Cesa-Bianchi, N. a., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*(2), 235–256.

Brügmann, B. (1993). Monte carlo go. `http://www.cgl.ucsf.edu/home/pett/go/Programs/Gobble.html`.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In van den Herik, J. H., Ciancarini, P., & Donkers, j. H. H. L. M. (Eds.), *Proceedings of the 5th International Conference on Computer and Games*, Vol. 4630/2007 of *Lecture Notes in Computer Science*, pp. 72–83, Turin, Italy. Springer.

Dellaert, F., Fox, D., Burgard, W., & Thrun, S. (1999). Monte carlo localization for mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA99)*, pp. 3–4.

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, *7*, 1–30.

Gelly, S., & Wang, Y. (2006). Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*.

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). Modification of UCT with patterns in Monte-Carlo Go. Tech. rep. 6062, INRIA, France.

Nash, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, *36*(1), 48–49.

Pengelly, J. (2002). Monte carlo methods. `http://www.cs.otago.ac.nz/cosc453/`. Accessed 2009-March-18.

Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *11*, 7–10.

Sensei's Library (2000a). Ko fighting. http://senseis.xmp.net/?KoFighting. Accessed 2009-March-08.

Sensei's Library (2000b). Suicide. http://senseis.xmp.net/?Suicide. Accessed 2009-March-08.

van der Steen, J. (2002). 9x9, book 4. http://www.gobase.org/9x9/book4/. Accessed 2009-April-29.

Zobrist, A. L. (1970). A new hashing method with application for game playing. Tech. rep. 88, Computer Science Department, The University of Wisconsin.