Fachbereich Mathematik und Informatik
Arbeitsgruppe Künstliche Intelligenz
Freie Universität Berlin

Bachelorarbeit

Monte-Carlo Methoden für das Spiel Go

Autor: Daniel Wäber
Betreuer: Prof. Dr. Raúl Rojas, Dr. Marco Block-Berlitz

Januar 2010

Zusammenfassung

Trotz der wenigen Regeln stellt das Spiel Go eine große Herausforderung an die Künstliche Intelligenz dar. Ein neuer Ansatz zur Planung komplizierter Vorgänge, die Monte-Carlo-Baumsuche, brachte erste Erfolge im Go [11], dennoch sind die besten Go-Computer noch weit vom Meister-Niveau entfernt.

Diese Arbeit setzt sich mit dem Verfahren im Kontext des 9×9 Go Spiels auseinander und soll eine Basis für weitere Untersuchungen, Verbesserungen und Erweiterungen der Monte-Carlo-Baumsuche legen. Die Ansätze werden diskutiert, eine Implementation der zugrundeliegenden Suche aufgezeigt und verschiedene Heuristiken und Erweiterungen der Suche vorgestellt. Durch Testreihen werden die Ansätze evaluiert.

Abbildung 1: ,,kifoo" – Logo der implementierten Go-KI

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, die Bachelorarbeit selbstständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Daniel Wäber, Berlin den 27.01.2010

Inhaltsverzeichnis

Danksagung

Ich will mich bei Prof. Dr. Rojas für die Annahme der Arbeit und für seine lehrreichen und spannenden Vorlesungen bedanken.

Besonderer Dank geht an Dr. Marco Block-Berlitz für die Unterstützung beim Erstellen der Arbeit, sowie an die Spiele-KI Gruppe der FU-Berlin. Desweiteren will ich mich bei Peggy Sabri, Benjamin Lorenz und Lars Kastner für ihre Anregungen und Korrekturen bedanken. Mein Dank geht auch an spline, deren Infrastrukturen für studentische Projekte auch bei dieser Arbeit hilfreich war.

Schließlich will ich mich noch bei den vielen Go-Spielern und der c-base bedanken, für die schönen Go-Partien und langen Nächte in gemütlicher Atmosphäre, die Go für mich zu einem ganz besonderen Spiel machten.

Kapitel 1

Motivation und Einführung

Erst vor kurzem gelang es einem Go-Computer das erste Mal einen hochrangigen Profi-Spieler in der vereinfachten 9×9 Go Variante zu besiegen. Selbst bei der vereinfachten Variante gingen diesem Sieg viele Niederlagen voraus. Auf dem 19×19 Go-Brett haben selbst die besten Go-Computer wenig Chancen gegen gute Amateure.

Im Vergleich zu anderen Brett-Spielen wie Schach muss sich die KI im Go noch stark verbessern, damit sie Meister-Niveau erreicht: Der hohe Verzweigungsgrad, die Schwierigkeit einer guten Evaluation und der Einfluss von lokalen Kämpfen auf eine globale Strategie stellen für die KI-Programmierung eine große Herausforderung dar.

Der neue Ansatz, die Monte-Carlo-Baumsuche, konnte erste Erfolge im Go erzielen. Die Weiterentwicklung des Ansatzes ist vielversprechend, dennoch müssen auch hier noch einige Hürden überwunden werden.

1.1 Aufbau der Arbeit

Nach einer kurzen Einführung in die Grundbegriffe des Go-Spiels folgt im Anschluss ein Überblick über verschiedene Ansätze und Arbeiten, die sich mit Baumsuche, Monte-Carlo-Methoden und dem Go-Spiel beschäftigen (Kapitel 2). Es werden sowohl klassische Suchverfahren auf Spielbäumen, wie auch verschiedene Monte-Carlo-Methoden vorgestellt. Diese werden in dem neuen Ansatz der Monte-Carlo-Baumsuche zusammengeführt. Der UCT-Algorithmus, der die Grundlage für diesen Ansatz bildet, sowie verschiedene Erweiterungen und Verbesserungen werden behandelt.

Die darauf folgenden Abschnitte befassen sich mit der Implementation. Zuerst wird eine Übersicht über die Struktur des Gesamtprojekts gegeben und der Ablauf und die Steuerung des Programms vorgestellt (Kapitel 3). Anschließend werden die Herzstücke der Go-KI genauer behandelt – die Spielbrett-Repräsentation (Kapitel 4) und die Baumsuche mit ihren Erweiterungen (Kapitel 5).

Schließlich werden die Ergebnisse vorgestellt (Kapitel 6) und ein Ausblick auf zukünftige Entwicklungen gegeben (Kapitel 7).

1.2 Das Go Spiel

Go oder weiqi (圍棋), wie es auf Chinesisch heißt, ist ein Jahrtausende altes Brettspiel und bedeutet wörtlich „Umschließungs-Spiel". Ziel ist es, möglichst viel Gebiet auf dem Spielbrett, dem Goban, mit seinen Steinen zu umschließen.

1.2.1 Regeln und Aufbau

Durch vier Regeln werden die Gesetzmässigkeiten des Spiels beschrieben:

- Zwei Spieler setzen abwechselnd schwarze und weisse Spielsteine auf die Schnittpunkte der Linien des quadratischen Spielbretts.
- Steine gleicher Farbe, die aneinander angrenzen, bilden ein Steinkette. Leere Felder, die an die Steinkette angrenzen werden als Freiheiten der Gruppe bezeichnet.
- Wird eine Steinkette von Steinen der gegnerischen Farbe komplett umschlossen, d.h. der Kette wird die letzte Freiheit genommen, so wird diese geschlagen und vom Brett genommen. Sie ist gefangen worden.
- Eine weitere Regel, die Ko-Regel, verhindert endlose Wiederholungen, wenn sich zwei Steine gegenseitig immer wieder schlagen könnten. Der Zug, der wieder zu der gleichen Situation führt, ist verboten.

Abbildung 1.1: Ein Goban

Das Spiel ist zu Ende, wenn beide Spieler keinen sinnvollen Zug finden und passen. Um den Gewinner des Spiels zu ermittelt, muss die Größe des Gebiets der beiden Spieler verglichen werden. Es gibt verschiedene Zählweisen, die allerdings bis in wenige Ausnahmen zum gleichen Ergebnis führen.

In den chinesischen Regeln wird die Grösse des Gebiets eines Spielers als die Anzahl der Schnittpunkte definiert, die von seinen Steinen umschlossen oder belegt sind.

Aus diesen einfachen Regeln folgen komplexe Abhängigkeiten und Konsequenzen, die Go zu einem faszinierenden Strategiespiel machen. Unter `http://playgo.to/iwtg/en/` ist eine interaktive Einführung in das Go-Spiel zu finden, die die Regeln und deren Konsequenzen verdeutlicht.

1.2.2 Taktische Folgerungen und Begriffe

In diesem Abschnitt werden einige Fachbegriffe des Go-Spiels erläutert, die für die Entwicklung einer Go-KI von besonderer Bedeutung sind.

Leben und Tod

Es ist möglich, Steinkonstellationen aufzubauen, die nicht mehr gefangen werden können. Ein solche Konstellation wird als lebending bezeichnet.

Dagegen gilt eine Gruppe von Steinen als tot, falls nicht mehr verhindert werden kann, dass der Gegner die Gruppen im Verlauf des Spiels fangen kann.

Augen

Einzelne Freiheiten, die von einer Steingruppe komplett umschlossen werden, werden als Augen der Gruppe bezeichnet. Besitzt eine Gruppe mindestens zwei Augen, so ist sie stabil: Die Augen stellen zwei Freiheiten der Gruppe da, die der Gegner niemals Gleichzeitig besetzen kann, da dessen Steine im Auge selbst keine Freiheiten besitzen und somit nach dem Zug vom Brett genommen werden.

Eine Gruppe mit zwei Augen ist also ein Beispiel für eine lebendige Konstellation. Eine Gruppe, die nicht genügend Freiraum hat, um zwei Augen zu bilden, ist dagegen tot.

Atari

Hat eine Steinkette nur eine Freiheit, so wird dies mit Atari bezeichnet. Eine sofortige Reaktion auf diese Situation ist nötig, um die Steinkette zu retten.

Leichtigkeit und Stärke

Eine Gruppe gilt als schwach bzw. stark, wenn das Leben der Gruppe mit wenigen bzw. nur mit vielen Steinen bedroht werden kann. Schwache Steingruppen, bei denen die Steine dicht zusammen liegen, werden als schwer bezeichnet, lose Einstreuungen von Steinen gelten als leicht.

Formen

Viele der entstehenden Formen auf dem Brett wurden benannt (z.B. Bambus-Verbindung oder Drachenmaul) und haben taktische und strategische Folgen.

Fuseki und Joseki

Wie auch beim Schach, haben sich gewisse Standard-Abspiele etablieren können. Diese werden vor allem in der Eröffnung (Fuseki) benutzt, aber auch in Früh- und Mittelspiel-Auseinandersetzungen (Joseki). Dennoch sind diese stets nur als Grund-Varianten aufzufassen und müssen an die jeweilige Situation angepasst werden.

Für einen Go-Spieler ist sowohl das genaue ,,lesen", also das Vorausplanen möglicher Abfolgen, wichtig, als auch eine ,,Intuition" für gute Formen und wichtige Punkte. Es muss das Ergebnis verschiedener lokaler Kämpfe vorausschauend bestimmt werden, aber zugleich die lokalen Möglichkeiten zu einem globalen Gesamtbild zusammen gesetzt werden.

Für eine künstliche Intelligenz stellen all diese Aufgaben eine große Herausforderung dar. Das Zusammensetzen von Teillösungen zu einem schlüssigen Handeln bildet eine weitere Schwierigkeit. Noch gibt es keinen Ansatz, der all diese Probleme lösen kann.

Kapitel 2

Verwandte Arbeiten

In diesem Kapitel werden wichtige Methoden der Spiele-KI vorgestellt, auf denen diese Arbeit aufbaut:

- Eine Grundlage für das vorausschauende Planen bei vielen Spielen sind Algorithmen auf einem Suchbaum wie das Minimax-Verfahren.
- Monte-Carlo-Verfahren konnten mit Erfolg zur approximativen Lösung von schweren Problemen verwendet werden und hielten deshalb auch Einzug in die Spiele-KI.
- Der UCT-Algorithmus, dessen Namen sich von ,,Upper Confidence Bound applied to Trees" herleitet, verbindet Suchbaum-Verfahren mit Monte-Carlo-Methoden und konnte im Go erfolgreich angewandt werden.

2.1 Suchbaum-basierte Verfahren

Eine wichtige Abstraktion zur Modellierung der Suchstrategie für Zweispieler-Nullsummenspiele stellt der Suchbaum dar.

Hierbei handelt es sich um einen Baum dessen Knoten eine Spiel-Situation und dessen Kanten mögliche Züge repräsentieren. Die Wurzel des Baumes repräsentiert die aktuelle Situation. Ziel eines Such-Verfahrens ist es, den Zug zu finden, der für den aktuellen Spieler zu einem bestmöglichen Ergebnis führt, wobei sich bestmöglich auf die optimale Spielweise des Gegners bezieht.

Ein wichtiges, grundlegendes Verfahren für die Suche des optimalen Zuges stellt der Minimax-Algorithmus dar. Die auf dem Minimax-Verfahren aufbauenden Algorithmen wie α-β-Suche, SSS* oder Negamax konnten in vielen Spielen erfolgreich angewandt werden [4].

2.1.1 Minimax-Verfahren

Um die optimale Lösung in einem Suchbaum eines Zweispieler-Nullsummen-Spiels zu bestimmen, wird das Ergebnis eines Knotens bestimmt, indem das aus dem Blickwinkel des handelnden Spielers bestmögliche Ergebnis der nachfolgenden Knoten übertragen wird. Ausschnitt 1 zeigt Pseudocode für den Minimax-Algorithmus.

Ausschnitt 1 Minimax-Algorithmus

```
Input: node
Output: minmax value of node
  if node is a leaf then
    return evaluation of node
  end if
  best ← −∞
  for all child of node do
    eval ← minmax value of child
    eval ← −eval // good for other play is bad for current play
    if eval > best then
      best ← eval
    end if
  end for
  return best
```

Durch diese Auswahl der besten Knoten entsteht ein Pfad im dem Spielbaum, die sogenannte Hauptvariante. Diese führt zum bestmöglichen Ergebnis aus Sicht beider Spieler. Weicht einer der Spieler von der Hauptvariante ab, kann sich sein Ergebnis nur verschlechtern.

Verbesserungen wie die α-β-Suche schneiden Äste ab, die garantiert zu schlechteren Ergebnissen für eine Situation führen. Damit verringert sich der Rechenaufwand, wobei dennoch garantiert werden kann, dass die Hauptvariante gefunden wird.

2.1.2 Evaluation

Bei komplexeren Spielen wie Schach oder Othello reichen die Ressourcen (Speicher und Zeit), die uns zur Verfügung stehen, nicht aus, um diese bestmögliche Lösung im Spielbaum zu finden. Deshalb muss die Suche vereinfacht werden.

Eine Möglichkeit, um Minimax-Verfahren dennoch anwenden zu können, ist es, die Suche nicht bis zum Ende des Spieles durchzuführen, sondern nach einer gewissen Anzahl von Zügen (dem Horizont) das Ergebnis für die Spiel-Situation durch eine Bewertungsfunktion zu nähern. Die Bewertungsfunktion sollte möglichst das Ergebnis liefern, das bei optimalem Spiel beider Spieler ab dieser Situation entsteht.

Zur Evaluation einer Situation wird meist Mustererkennung und Feature-Extraktion in Verbindung mit maschinellem Lernen benutzt.

2.1.3 Transposition

Bei vielen Spielen, so auch im Go, können identische Brett-Situationen über verschiedene Zugfolgen erreicht werden. Wird eine solche Transposition im Suchbaum identifiziert, muss diese Situation nur für eine der möglichen Zugfolgen bewertet werden.

Da im Go-Spiel fast jeder Schnittpunkt ein legaler Zug ist, der nur lokale Veränderungen zur Folge hat, treten häufig Transpositionen auf. So konnte z.B. beim Lösen des 5×5 Go durch das Einsetzen der Transpositions-Tabelle der Spielbaum um mehr als 99% reduziert werden [16].

Zur Identifikation von Transpositionen kann eine Transpositions-Tabelle benutzt werden. Dies ist eine spezielle Hash-Tabelle, in der für bereits bekannte Situationen deren Bewertung gespeichert wird. Anstatt der aufwendigen Berechnung der Bewertung kann bei einer Transposition der Wert aus der Tabelle benutzt werden. Um damit die Effizienz der Suche zu steigern ist es besonders wichtig, dass die benutzte Hash-Funktion schnell berechenbar ist, aber wenige Konflikte und eine gute Verteilung besitzt.

2.1.4 Zobrist-Hash

Für verschiedene Brettspiele, wie auch Go, eignet sich für die Berechnung eines Hash die Methode nach Zobrist [17]. Dazu wird für jede Position und jede Spielfigur ein zufälliger Bit-String der Länge k generiert. Der Hash-Wert h einer Spielsituation ist ebenfalls ein Bit-String der Länge k. Dieser wird inkrementell mit jeder Aktion am Spielbrett aktualisiert. Wird eine Figur f von einer Position p wegbewegt oder auf eine Position p gesetzt, so wird h wie folgt erneuert:

$$h \leftarrow h \oplus bitstring(f, p) \tag{2.1}$$

Zusätzlich müssen Zusatzinformationen, die die Bewertung verändern, im Hash kodiert werden. Das sind z.B. der aktuelle Spieler oder gegebenenfalls eine Ko-Position. Hierfür können weitere Bit-Strings, die zu dem Hash addiert werden, oder zusätzliche Bits im Hash verwendet werden.

Die inkrementelle Berechnung mit einer einfach Operation ist sehr schnell und durch die zufällige Initialisierung und die Eigenschaften der *xor*-Operation ist eine gute Verteilung garantiert. Über die Grösse der Schlüssel kann die Fehlerwahrscheinlichkeit reguliert werden. Bei einer maximalen Tabellengrösse von 2^n genügen schon $n + 10$ *bit*, um die Wahrscheinlichkeit eines Fehler kleiner als $0,1\%$ zu halten [17]. Ein 64bit Hash reicht also bei weitem aus, um die Möglichkeit einer Kollision in dem generierten Spielbaum ignorieren zu können.

2.2 Monte-Carlo-Methoden

Zur approximativen Lösung von komplexen physikalischen und mathematischen Problemen konnten Monte-Carlo-Methoden erfolgreich angewandt werden.

In den 1990ern wurden Monte-Carlo-Methoden erstmals bei Spielen mit perfekter Information verwendet [3] und 1993 entwickelte Bernd Brügmann das erste Go Programm, das auf Monte-Carlo-Methoden basiert [8]. Anstatt die verbreiteten Methoden einer auf Experten-Wissen aufbauenden Evaluation mit lokaler α-β-Suche zu verbessern, versuchte Brügmann eine Methode zu entwickeln, die für den Computer ,,natürlicher ist":

- Beginnend mit der Ursprungssituation werden solange Spielzüge mit einem zufallsbasierten Verfahren ausgewählt und angewandt, bis eine terminale Position erreicht ist.
- Diese Position kann leicht evaluiert werden, indem der Sieger bestimmt wird.
- Die ersten beiden Schritte werden wiederholt und schließlich derjenige Zug zurückgegeben, der im Mittel am besten abschneidet.

Ohne jegliches Experten-Wissen erreichte Brügmanns Computer-Spieler Gobble schon besseres Anfänger-Niveau. Durch die Veränderung der zufallsbasierten Zug-Auswahl konnte sich Gobble sogar auf 12kyu steigern [7], ein Rang, den begabte Go-Spieler nach ca. einem Jahr erreichen.

In den folgenden Abschnitten werden verschiedene Verfahren zur Zug-Auswahl und zusätzliche Heuristiken vorgestellt, die erfolgreich in Monte-Carlo basierten Go-KIs angewandt werden.

2.2.1 Sampling zur Bewertung

Die Verwendung von Monte-Carlo-Methoden in nicht-zufallsbasierten Spielen baut auf dem Modell des ,,Expected Outcome" von Abramson auf [3]:

Zur Bewertung von Blatt-Knoten wird das Spiel mit zufällige Zügen zu Ende geführt (Monte-Carlo-Sampling oder auch Playout genannt), um schließlich aus mehreren solchen Spielen den erwarteten Ausgang für den Knoten zu bestimmen (siehe Algorithmus 2). Diese Heuristik wendete Abramson auf Othello an und schließt, diese Methode zur Evaluierung sei "precice, accurate, easily estimable, efficiently calculable, and domain independent" [3].

Ausschnitt 2 Monte-Carlo-Evaluation

Input: *state*, *#samples*
Output: evaluation of *state*

```
wins ← 0
games ← 0
while games ≤ #samples do
  state′ ← copy of state
  while state′ is not terminal do
    apply random action on state′
    // note that the actions don't have to be pure random,
    // but can be filtered and biased to improve the evaluation
  end while
  games ← games + 1
  if actor of state equals winner of state′ then
    wins ← wins + 1
  end if
end while
return wins/games
```

Die Bewertung von Zügen durch Monte-Carlo-Sampling begründet die Verwendung von Monte-Carlo-Methoden im Go und stellt die Grundlage für den Erfolg der aktuellen Go-Programme dar.

2.2.2 Simulierte Abkühlung und Temperatur

Die Methode der Simulierten Abkühlung ist eines der wichtigsten Monte-Carlo-Verfahren zur Bestimmung eines globalen Extremwertes.

Das Verfahren kann durch das Bild des abkühlenden Stahls in einer Schmiede verdeutlicht werden [8]: Ziel beim Schmieden ist es, eine perfekte Anordnung der Moleküle zu erreichen (Extremwert). Durch das Erhitzen das Stahls wird eine

zufällige Bewegung der Atome hervorgerufen (Monte-Carlo-Verfahren). Kühlt der Stahl ab, ordnen sich die Atome energetisch bestmöglich an. Dabei ist darauf zu achten, dass die Abkühlung langsam erfolgt. Bei schneller Abkühlung würden die Bewegung der Atome nur ausreichen, ein lokales Minimum zu erreichen bevor der Stahl erhärtet. Durch einen langsamen Energieverlust nimmt die Wahrscheinlichkeit ab, dass die Atome energetisch schlechtere Anordnungen einnehmen, denn es können durch die Restbewegung der Atome lokale energetische Extrema überwunden werden.

Brügmann benutzte diese Methode für die Go-KI Gobble [8]. Wie unter Abschnitt 2.2 beschrieben, werden viele zufällige Spiele ausgeführt. Dabei wird die Zugauswahl von einer Temperatur-Variable beeinflusst, die im Verlauf der Berechnung langsam gesenkt wird:

- bei hoher Temperatur werden die Spielzüge rein zufällig ausgewählt,
- bei geringer Temperatur werden Züge bevorzugt, die eine hohe Durchschnittsbewertung in den bisher simulierten Spielen haben.

Dadurch soll erreicht werden, dass nach dem Abkühlen nur die Züge der Hauptvariante ausgeführt werden.

Bouzy, der verschiedene Monte-Carlo-Methoden für Go testete, stellte jedoch fest, dass eine konstante Temperatur, also die Züge durchwegs randomisiert auszuführen, keine signifikante Veränderung gegenüber einer Abkühlung zur Folge hat.

Bouzy schließt, dass die Tendenz zur Hauptvariante durch das Verfahren nach Brügmann nicht erreicht werden konnte. Eine mögliche Erklärung sieht Bouzy darin, dass Brügmann bei der simulierten Abkühlung nicht nur den jetzigen Zustand, also die Zugreihenfolge benutzt, sondern auch bisherige Ergebnisse der Simulationen durch die Bestimmung des Durchschnittswertes mit einschließt [5]. Ein weiteres Problem könnte die zu schnelle Abkühlung relativ zu der Komplexität des Go-Spiels darstellen, sodass Gobble meist nur einen Zug bestimmt, der ein lokales Minimum darstellt.

2.2.3 Flache und Baum-basierte Suche

Bei einer flachen Monte-Carlo-Suche wird jeder mögliche Zug durch Monte-Carlo-Sampling bewertet und der beste zurückgegeben. Auch wenn mit dieser Methode nicht vorausschauend geplant werden kann, führt sie zu einer KI die oft ,,intuitiv" einen guten Zug wählt.

Diese Methode gleicht der Simulierten Abkühlung mit konstanter Temperatur bzw. einem Suchbaum der Tiefe 1 mit Monte-Carlo-Bewertung nach Abramson.

Die Monte-Carlo-Bewertung als Evaluation einer globalen Minimax-Suche einzusetzen stellte sich für Go, wegen des hohen Verzweigungsgrades, jedoch als unpraktikabel heraus. Bouzy testete dies mit Tiefe 2, stellte aber eine Verschlechterung des Ergebnisses der Baumsuche gegenüber der flachen Bewertung fest, da er die Anzahl der Samples pro Blatt zu stark einschränken musste [5].

2.2.4 AMAF-Heuristik

Wie bei der Transposition im Spielbaum (2.1.3), kann auch das Ergebnis einer Monte-Carlo-Simulation, also die Endposition des zufallsbasierten Spiels, durch

verschiedene Spielzugfolgen erreicht werden. Das Ergebnis ist also für mehrere Zugfolgen gültig.

Auf dieser Tatsache beruht die All-Moves-As-First-Heuristik [5]: Statt das Ergebnis nur für den Anfangszug der zufällig ausgewählten Zugfolge zu berücksichtigen, wird es für alle Anfangszüge von Zugfolgen berücksichtigt, die zu derselben Endposition führen. Somit wird mit einer Simulation Information über mehrere Züge gleichzeitig gewonnen. Dies führt zu dem in Ausschnitt 3 gezeigten Algorithmus.

Ausschnitt 3 AMAF-Evalutation

```
Input: state, #samples
Output: win statistics of actions
  wins ← map form action to number
  games ← map form action to number
  firstmoves ← map from action to actor
  while games ≤ #samples do
    state′ ← copy of state
    while state′ is not terminal do
      action ← select random action of state
      apply action on state′
      if firstmoves does not contain action then
        firstmoves[action] = actor of state′
      end if
    end while
    for action in firstmoves where firstmoves[action] is actor of state do
      games[action] ← games[action] + 1
      if actor of state equals winner of state′ then
        wins[action] ← wins[action] + 1
      end if
    end for
  end while
  return wins, games
```

Bei einem Sampling-Verfahren mit gleichverteilter Auswahl der Züge ist das Ergebnis der AMAF-Heuristik äquivalent zu einer einfachen Simulation für jeden der Anfangszüge. Fließt in das Sampling-Verfahren Go-Wissen mit ein oder werden Ergebnisse vorheriger Simulationen verwendet, würde nicht jede der Zugfolgen, die zu dem Ergebnis führen auch für andere Anfangszüge gewählt werden. Die AMAF-Heuristik führt hier also zu einem anderem Ergebnis als eine Simulation der einzelnen Züge. Die zusätzliche Information kann allerdings hier immer noch genutzt werden, z.B. als schnellere erste Abschätzung, aber sie ist dann nicht mehr gleichbedeutend mit dem Ergebnis aus einzelnen Simulationen.

2.2.5 Progressiv-Pruning

Wie auch die Minimax-Suche, kann eine Monte-Carlo basierte Suche durch das Abschneiden schlechter Züge optimiert werden. Dazu wird die statistische Gleichheit und Ungleichheit zweier Züge M_1 und M_2 betrachtet:

Sei $M.m$ der Durchschnitt der Ergebnisse für einem Zug M, so definiert man den rechten und linken Erwartungswert $m_l = m - r_d\sigma$ und $m_r = m + r_d\sigma$

des Zuges, wobei r_d eine durch Experimente bestimmte Konstante und σ die Standardabweichung ist. Ein Zug M_1 ist statistisch schlechter als M_2, wenn $M_1.m_r < M_2.m_l$. M_1 und M_2 sind statistisch gleich, falls weder M_1 statistisch schlechter als M_2 oder M_2 statistisch schlechter als M_1 ist [5].

Basierend auf der Annahme der Unabhängigkeit der Monte-Carlo-Samples untereinander, können somit alle Züge, die statistisch schlechter sind als der jetzige beste Zug von der Suche ausgenommen werden.

Bouzy stellt durch diese Optimierung eine Verschlechterung der Spielstärke, aber einen massiven Zeitgewinn fest, der über den Parameter r_d gesteuert werden kann.

2.2.6 Expertenwissen und maschinelles Lernen

Obwohl bei Monte-Carlo-Verfahren der Wert einer Situation erst durch viele Samples gewonnen wird, können auch hier Expertenwissen und Mustererkennung in Verbindung mit maschinellem Lernen das Ergebnis verbessern.

Eine Möglichkeit besteht darin, das Sampleverfahren zu verbessern. Indem Muster aus Profispielen extrahiert wurden und Züge, die solche Muster erzeugen, in der Zugauswahl bevorzugt wurden, konnte Brügmann die Spielstärke von Gobble deutlich steigern [7].

Bei Ansätzen zum maschinellen Lernen sollte dabei nicht die Spielstärke des Zufallsspielers optimiert werden, sondern der Fehler der Evaluation, die aus den Zufallsspielen resultiert. Dazu dient das Verfahren des Balanced Learning [6], das nicht einzelne Ergebnisse optimiert, sondern den Durchschnitt vieler Zufallsspiele an eine Referenz-Evaluation anpasst.

2.3 Monte-Carlo-Baumsuche

Für Markow-Entscheidungsprozesse mit großen Zustandsräumen können Monte-Carlo-Ansätze ebenfalls effizient fast-optimale Lösungen finden. Die Zusammenführung von Baumsuche, dem Sampling zum Bewerten von zufallsbasierten Spielen und der Monte-Carlo-Ansätze zum Lösen von Markow-Entscheidungsprozessen führte zu einem neuartigen Ansatz, der erfolgreich bei Go eingesetzt wurde.

2.3.1 Markow-Entscheidungsprozess

In einem Markow-Entscheidungsprozess (MDP) wählt der Akteur in einem Zustand S eine Aktion a. Abhängig von dem aktuellen Zustand S und der Wahl der Aktion a tritt der nächste Zustand S' mit einer Wahrscheinlichkeit $P_a(S, S')$ ein und der Akteur erhält eine Belohnung $R_a(S, S')$. Dabei hängt $P_a(S, S')$ und $R_a(S, S')$ nur von S und a ab und ist unabhängig von allen vorangegangen Zuständen [1].

Das Go-Spielbrett kann also als Zustand eines Entscheidungsprozess gesehen werden. In einem Zustand S folgt auf eine Aktion a allerdings immer der gleiche Folgezustand S', $P_a(S, S')$ ist also nur für diesen Zustand S' gleich 1 und die direkte Belohnung $R_a(S, S')$ ist meist 0, nur für Endzustände kann der Sieger bestimmt werden und somit eine Belohnung für den Sieg zurückgegeben werden.

2.3.2 k-armiger Bandit

Das k-armige Banditen-Problem beruht auf der Vorstellung von Spielautomaten mit mehreren Hebeln. Jeder Hebel hat eine feste Belohnungsverteilung, die dem Spieler allerdings unbekannt ist. Ziel des Spielers ist es, seinen Gewinn zu maximieren.

Dabei tritt das sogenannte „Exploration-Exploitation"-Dilemma zu Tage [13]: Sollte der Spieler den Hebel betätigen, der ihm von den bisherigen Ergebnissen am besten erscheint, oder soll er Hebel ausprobieren, die er wenig gespielt hat, bei denen er sich also nicht sicher über das Ergebnis ist?

Eine Lösung des Problems ist die Upper-Confidence-Bound-Strategie. Es wird also stets derjenige Hebel i gespielt, der die höchste obere Schranke c_i für den erwarteten Gewinn besitzt [15]:

$$c_i = \bar{x}_i + \sqrt{\frac{2 \log n}{n_i}} \tag{2.2}$$

wobei $\bar{x}_i$ der durchschnittliche Gewinn ist, n die Anzahl der Spiele insgesamt und n_i die Anzahl der Spiele an dem Hebel i ist. Ausschnitt 4 verdeutlicht die Auswahl nach UCB und wird im UCT-Algorithmus (siehe 2.3.3) wiederverwendet.

Ausschnitt 4 UCB Selektion

```
Input: list of actions with properties x̄_i, n_i
Output: action with highest UCB
  n ← 0
  for i in list do
    n ← n + n_i
  end for
  val ← −∞
  best ← void
  for i in list do
    if n_i == 0 then
      v = ∞
    else
      v ← x̄_i + sqrt(2 log n / n_i)
    end if
    if v > val then
      val ← v
      best ← action i
    end if
  end for
  return best
```

2.3.3 UCT-Algorithmus

Der UCT-Algorithmus ist ein Monte-Carlo-Algorithmus, mit dessen Hilfe der Akteur eine Bewertung für die Aktionen in einem MDP mit grossem Zustandsraum lernt.

Hierzu wird die UCB-Strategie angewandt, um den Monte-Carlo-Sampling-Vorgang in dem MDP zu leiten: Da nicht der komplette Zustandsraum besucht werden kann, ist es sinnvoll, in jedem Zustand diejenigen Aktionen genauer zu untersuchen, die bis dahin am vielversprechendsten sind. Ein MDP und auch Go kann also wie ein mehrstufiges Banditen-Problem behandelt werden [15].

Dazu wird ein Baum aufgebaut, dessen Knoten einen Zustand und dessen Kanten die Übergänge durch Ausführen einer Aktion repräsentieren. Zusätzlich wird die Anzahl der Besuche n_i und der durchschnittliche Gewinn $\bar{x}_i$ in den Knoten gespeichert. Es reicht aus, nur öfters besuchte Knoten, also solche mit $n_i > 0$, im Baum zu speichern, Knoten die noch nicht im Baum vorhanden sind, können als Knoten mit $n_i = 0$ und $\bar{x}_i = 0$ behandelt werden. Durch das Besuchen der Knoten wächst der Baum langsam von der Wurzel aus.

Ausschnitt 5 fasst die Schritte des UCT-Algorithmus zusammen. Dabei kann zwischen der Baum-Phase, in dem Knoten nach dem UCB-Verfahren ausgewählt werden (siehe Algorithmus 4 zur UCB-Selektion) und der Sampling-Phase, in der zufällige Aktionen ausgewählt werden (siehe Algorithmus 2) unterschieden werden. Die dritte Phase führt das Ergebnis des Samples in den Baum zurück.

Ausschnitt 5 UCT Algorithmus

Input: *state*, *root*
Output: best action
  while time left > 0 **do**
    $nodes[0] \leftarrow root$
    reset *state* to state of *root*
    $i \leftarrow 0$
    while $nodes[i].n \geq 0$ **do**
      $a \leftarrow$ select action from $nodes[i]$ according to ucb-selection
      apply a on *state*
      $nodes[i+1] \leftarrow$ child with action a of $nodes[i]$
      $i \leftarrow i + 1$
    end while
    sample *state* randomly
    eval $\leftarrow$ evaluation of *state*
        (for Go: eval $\leftarrow 1$ *for a win, -1 or 0 for an loss)*
        (for MDP: calculate the total reward for each actor)
    for all $node \in nodes$ **do**
      update $node.\bar{x}$ according to *eval* with the perspective of the *node*s actor
        (for Go: alternate eval and $-$*eval like in Minimax (2.1.1))*
      $node.n \leftarrow node.n + 1$
    end for
  end while
  return best action of *root*

Der Algorithmus kann jederzeit abgebrochen werden und liefert einen guten Näherungswert für die Lösung des MDP. Kocsis und Szepesvari zeigen, dass der Algorithmus zur optimalen Lösung konvergiert [15]. Im Falle des Spiel-Baums eines Zweispieler-Nullsummenspiels konvergiert der Algorithmus also zur Hauptvariante des Minimax-Baums.

2.4 Erweiterungen

Um die Effizienz des Algorithmus für Spielbäume und im Speziellen für Go zu steigern, können einige Erweiterungen und Verbesserungen angewandt werden, die sich teils aus bekannten Verfahren für Spielbäume und teils aus Verbesserungen der Monte-Carlo-Verfahren ableiten.

2.4.1 Parallelisierung

Der UCT-Algorithmus kann gut parallelisiert werden, um heutige Multi-Core-Architekturen auszunutzen. Dabei kann zwischen Wurzel-, Blatt-, und Baum-Parallelisierung unterschieden werden [9].

Wurzel-Parallelisierung

Ein einfacher Ansatz ist das parallele Ausführen mehrerer Threads, die jeweils ihren eigenen Baum verwalten und das Zusammenfuhren der Bäume am Ende. Allerdings wird hier die Rechenzeit nicht ideal ausgenutzt, da einzelne Threads oft die gleichen schlechten Züge besuchen müssen, bevor diese verworfen werden können.

Blatt-Parallelisierung

Eine weitere Möglichkeit ist eine serielle Ausführung der Baum-Phase und das parallele Ausführen der Sample-Phase. Hier werden die Ergebnisse ideal zusammengeführt, allerdings stellt der Zugriff auf den Baum einen Flaschenhals dar.

Baum-Parallelisierung

Da die einzelnen Ausführungen meist auf unterschiedlichen Teilen des Baums agieren, ist auch eine Parallelisierung in der Baum-Phase möglich. Die Veränderungen der Knoten-Werte n_i und $\bar{x}_i = \frac{wins_i}{n_i}$ kann durch atomare Operationen geschehen, sodass hierfür keine Locks notwendig sind. Wird der Zähler n_i eines Knotens schon in der Selektionsphase erhöht, kann vermieden werden, dass mehrere Threads gleichzeitig einen Pfad testen, bei dem die schon gestartete Simulation die Unwichtigkeit des Pfades zeigen könnte. Die Erweiterung eines Knotens kann entweder durch ein Lock geschützt werden, oder über eine Transaktion gesichert sein.

2.4.2 Rapid-Action-Value-Estimate

RAVE verwendet die AMAF-Heuristiken im UCT-Kontext, um schneller den Wert einer Aktion in einem Knoten approximieren zu können [12]. Dazu werden nicht nur die Gewinn-Statistiken in dem Knoten gespeichert, sondern auch für jeden der möglichen Züge die Anzahl der ,,virtuellen" Gewinne und Besuche aus den AMAF-Daten.

Diese Daten sind oft im Gegensatz zu den echten Gewinn-Statistiken verzerrt, da in der Baumsuche die Annahme der Unabhängigkeit nicht mehr gegeben ist. Allerdings liegt schon nach wenigen Besuchen eines Knotens eine gute Schätzung für dessen Wert vor. Deshalb werden in die Selektierung von wenig besuchten Knoten die AMAF-Statistiken stärker mit einbezogen, bei oft besuchten Knoten wird die unverzerrte Gewinn-Statistik benutzt.

2.4.3 Maschinelles Lernen

Die Möglichkeiten der Verbesserung der Monte-Carlo-Samples, wie in 2.2.6 beschrieben, können auch in der Sampling-Phase des UCT-Algorithmus angewandt werden.

Zusätzlich können über Mustererkennung und Expertenwissen Startwerte für $\bar{x}_i$ und n_i, bzw. für die entsprechenden RAVE-Werte gelernt werden, sodass Züge, die bekannte gute Stellungen erzeugen, bevorzugt getestet werden [2].

2.4.4 Transposition

Eine klassische Transposition, also das Zusammenführen von Knoten, die die gleiche Situation repräsentieren, ist im UCT-Algorithmus problematisch, da die Exploration über die Anzahl der Besuche eines Knotens gesteuert wird. So könnten Situationen auftreten, in denen beste Züge nicht gefunden werden.

Eine Möglichkeit, Transpositionen dennoch zu benutzten, ist es, nur die Durchschnittsbewertung $\bar{x}$ von Knoten, die der gleichen Situation entsprechen, zusammenzuführen. Allerdings kann damit das Ergebnis der Suche nur geringfügig gesteigert werden, da die Knoten dennoch auf verschiedenen Wegen besucht werden müssen und nur deren Bewertung etwas präziser ist [10].

Kapitel 3

Projekt-Übersicht

In dem Projekt kifoo wird die Monte-Carlo-Baumsuche mit verschidenen Heuristiken und Erweiterungen implementiert und evaluiert. Der Name kifoo spielt auf den japanischen Begriff 碁風 (kifuu - Go Spiel-Stil) sowie auf den Computer Slang `foobar` an – der Spiel-Stil des Go-Computer soll gefunden werden. Der Stil ist noch unbekannt (die beliebte Variable `foo`) und dessen Wert muss erst durch Untersuchungen bestimmt werden.

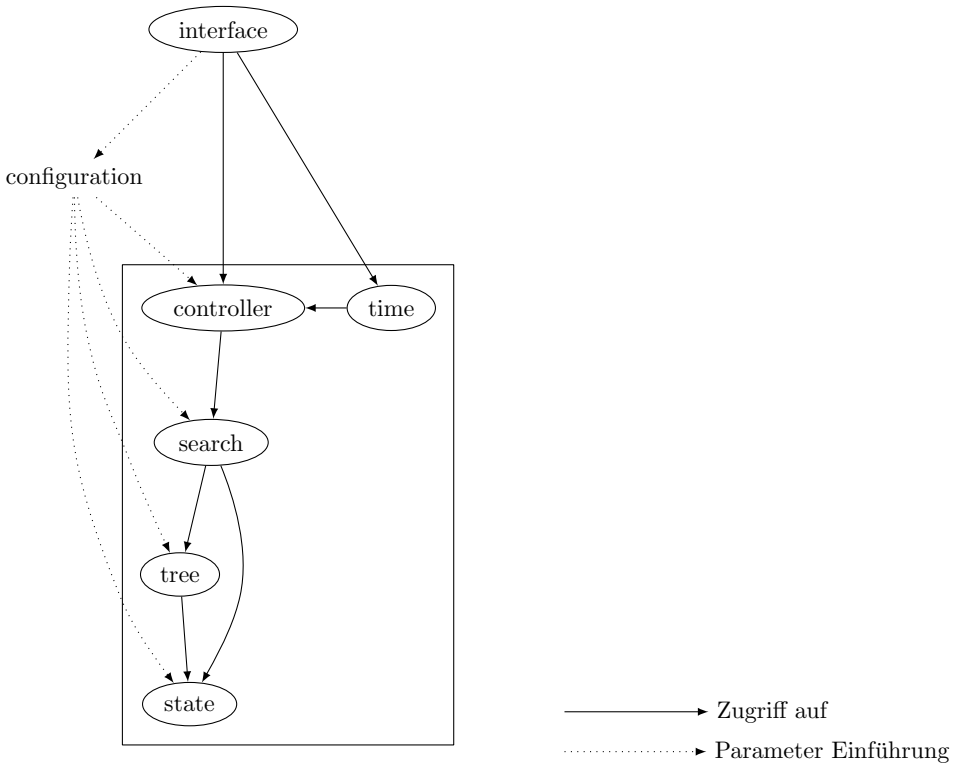

Abbildung 3.1: Struktur des Projektes

3.1 Architektur

3.1.1 Ziele

Ziel des Projektes ist die Implementierung eines Computer-Go-Spielers aufbauend auf einer "state of the art" Monte-Carlo Baumsuche. Die Einbettung herkömmlicher Spiele-KI Methoden und anderer Erweiterungen der Monte-Carlo Suche sollte einfach zu realisieren und testen sein.

Das Projekt lässt sich grob in die drei Bereiche, Steuerung , Such-Algorithmus und Spielbrett-Repräsentation einteilen.

Steuerung und Anbindung

Funktionale Ziele:

- Verwaltung der Suche
  - Spieler und Spielzustand
  - Zeitverwaltung
- Go Text Protocol zur Anbindung an
  - Computer Go Server und KGS[1] Go Server
  - andere Go-Computer
  - Go Regression Test-Suite

Nichtfunktionale Ziele:

- einfaches Testen verschiedener Konfigurationen

Such-Algorithmus

Funktionale Ziele:

- Monte-Carlo Baumsuche
- Implementation verschiedener Heuristiken
- Transpositionserkennung
- Parallelisierung

Nichtfunktionale Ziele:

- einfach erweiterbar
- Abstraktion, um Algorithmus auf andere Spiele anzuwenden

Spielbrett-Repräsentation

Funktionale Ziele:

- Zugerzeugung und Ausführung
- ergiebige Darstellung für Expertenwissen und Mustererkennung
- Zobrist-Hashfunktion
- Zustandsspeicherung/Austausch

[1] KGS Go Server – einer der größten Go Server, auf dem sowohl Menschen wie auch Go-Computer gegeneinander antreten können

Nichtfunktionale Ziele:

- schnelle Ausführung, um hohe Sample-Rate zu erlangen
- erweiterbar, um verschiedene Verfahren der Mustererkennung zu implementieren

3.1.2 Überblick

Die grobe Struktur leitet sich aus den Teilgebieten des Projektes ab: So verwaltet die Steuerung die Suche, startet diese, verwaltet die Zeit und fragt den besten Zug ab. Die Suche wiederum benutzt Spielbrett-Objekte, um auf diesen den Algorithmus auszuführen und Informationen über die besten Züge zu sammeln.

In seiner Studienarbeit kümmert sich Markus Decke um die Steuerung und die Anbindung über GTP[2]. Suche und Spielbrett-Repräsentation sind im Laufe dieser Arbeit entstanden. Thorsten Reinhard befasst sich in seiner Diplomarbeit mit Verfahren zur Mustererkennung und deren Anwendung in der Monte-Carlo-Suche.

Auf die Implementation der Spielbrett-Repräsentation sowie der Baumsuche und der verschiedenen Heuristiken wird genauer im am Anschluss an diese Kapitel eingegangen.

3.2 Infrastruktur

Verschiedene, verbreitete Technologien werden benutzt, die den Entwicklungsprozess beschleunigen und erleichtern, sowie die Verwaltung des Projektes unterstützen.

3.2.1 Web-Präsenz

Die Hauptseite (`https://dev.spline.de/trac/kifoo/wiki`) des Projektes ist ein von spline[3] verwaltetes Trac-Wiki. Dort befinden sich unter anderem:

- eine Einführung in das Projekt
- Informationen zur Ausführung des Programms
- Zugriff auf Quelltext und die Artefakte des Projektes
- verschiedene Anregungen zur Weiterentwicklung des Projektes
- Links zu allgemeinen Informationen über KI und Go
- eine Issue-Tracker zur Verwaltung von Bugs und anderen Aufgaben

Desweiteren befinden sich unter `http://page.mi.fu-berlin.de/waeber/kifoo/` Statistiken über den Quelltext des Projekt.

[2] Go Text Protocol – Protokoll zur Kommunikation von Go-Programmen (`http://www.lysator.liu.se/~gunnar/gtp/`)

[3] spline – Studentisches Projekt LInux Netzwerk (`http://www.spline.inf.fu-berlin.de/index.php/Hauptseite`

3.2.2 Ausführung

Das erzeugte oder heruntergeladene Jar-Package kann mit Java ausgeführt werden und kommuniziert mit Hilfe des Go-Text-Protokolls auf Standardein- und -ausgabe. Der dem Programm zur Verfügung stehende Speicher sollte mit `-Xms` und `-Xmx` erhöht werden:

```
java -server -Xms2g -Xmx2g -jar kifoo-1.0-SNAPSHOT-gtp.jar
```

Optionen können mit Hilfe weiterer Parameter eingestellt werden:

```
usage: kifoo <options>
 -e,--expand-threshold <int> expand-threshold: 2, # of
                             simulations before extending a
                             node, must be at least 1
 -a,--a <int>                a: 13, avoid bad luck constant
 -b,--b <int>                b: 17, avoid bad luck constant
 -f,--formula <string>       formula: noluck, formula to
                             calculate values, [noluck, ucb]
 -h,--help                   show this help message
 -p,--property               use property file to read settings
 -s,--simulations <int>      simulations: 50000, int
 -t,--threads <int>          threads: 2, int threads
 -u,--ucbk <float>           ucbk: 0.6, exploration factor
```

Als Endbenutzer-Schnittstelle kann z.B. Go-Gui[4] benutzt werden, das beliebige GTP-Programme steuert. Dort kann der Befehl zum Ausführen von kifoo eingetragen werden. Des Weiteren bietet Go-Gui die Funktionalität, mehrere Go-Computer gegeneinander antreten zu lassen:

```
./gogui-twogtp \
  -verbose \
  -size 9 \
  -time 10 \
  -games 10 \
  -auto \
  -alternate \
  -black 'gnugo --mode gtp' \
  -observer ./gogui-display \
  -referee 'gnugo --mode gtp' \
  -sgffile kifoo-test-`date +%y-%m-%d` \
  -white 'java -server -Xms2g -Xmx2g -jar \
          kifoo-1.0-SNAPSHOT-gtp.jar'
```

3.2.3 Quelltext und Kompilation

Das Projekt wurde mit Java realisiert, der Quelltext wird mit SVN[5] verwaltet und kann mit folgendem Kommandozeilen-Befehl heruntergeladen werden.

```
svn co https://dev.spline.de/svn/kifoo kifoo
```

[4] Go-Gui – Oberfläche für Go Programme (`http://gogui.sourceforge.net/`)
[5] Subversion – Versionsverwaltungssystem (`http://subversion.tigris.org/`)

Zum Build-Management wird Apache Maven[6] benutzt, welches die Verwaltung von Abhängigkeiten, die Kompilation, Tests und Packaging, sowie das Deployment automatisiert. Maven kann auf der Kommando-Zeile verwendet werden, aber es stehen auch für die gängige Entwicklungsumgebungen wie Netbeans und Eclipse Plugins zur Integration zur Verfügung. Ein jar-package kann z.B. mit mvn wie folgt erstellt werden:

```
cd kifoo
mvn package
```

3.2.4 Continuous Integration und Tests

Um Programmierfehler oder eine Verschlechterung der Go-KI schnell zu bemerken, werden JUnit- und Regression-Tests automatisiert ausgeführt sobald Quelltextänderungen in das SVN-Repository geladen werden. Dazu läuft unter `http://forti.ath.cx:8080/` ein Hudson[7] CI Server, der die Sources herunterlädt, Tests ausführt und Statistiken und Pakete erneuert. Falls Verschlechterungen auftreten, werden die Entwickler darauf hingewiesen.

Zusätzlich kann die KI mit Hilfe von einfachen Shell-Skripten verteilt auf verschiedenen Rechnern Testspiele gegen andere Go-Computer ausführen, sodass schnell Spielergebnisse zu bestimmten Konfigurationen oder Erweiterungen zur Verfügung stehen.

[6] Apache Maven – Werkzeug zur Projektverwaltung (`http://maven.apache.org/`)
[7] Hudson – Continuous Integration Server (`http://hudson-ci.org/`)

Kapitel 4

Spielbrett

Die Repräsentation des Spielbretts stellt die Funktionalität für Such- und Lernalgorithmen zur Verfügenung. Dabei müssen folgende Teilaufgaben behandelt werden:

Zustand speichern und laden, übergeben und zurücksetzen
zur Verwaltung der Suche und zur vorausschauenden Planung

Züge generieren und ausführen
für die Traversierung im Baum und die Zugwahl im Monte-Carlo-Playout

Repräsentation von taktischem Wissen und Mustererkennung
zur Verbesserung der Suche und als Grundlage maschinellen Lernens

Auswertung des Spiels
zur Rückführung des Ergebnisses in der Suche

4.1 Aufbau aus Komponenten

Zuggeneration, Repräsentation von taktischem Wissen, Mustererkennung und Auswertung können aber nicht auf eine festgelegte Implementation eingeschränkt werden. Verschiedene Ansätze müssen erst getestet werden und es muss stets mit einem Austausch oder einer Erweiterung einzelner Komponenten gerechnet werden. Dennoch sind einige Strukturen miteinander gekoppelt: So kann z.B. der Zuggenerator auf Merkmale der Mustererkennung zugreifen, um verbesserte Samples zu generieren.

Einzelne Teilgebiete der Spielbrettrepräsentation hängen also von anderen ab, müssen aber dennoch austauschbar bleiben. Da Such- und Lernalgorithmen viel mit dem Spielbrett interagieren, ist zusätzlich eine auf Geschwindigkeit ausgelegte Lösung wichtig.

Die Austauschbarkeit kann durch einen Aufbau des Spielbretts aus einzelnen Komponenten erreicht werden, die

- einen Dienst/eine Teilfunktionalität übernehmen
- andere Dienste von Komponenten nutzen können

Ein Dienst kann also einerseits einen Teil der Funktionalitäten des Spielbretts übernehmen, der Dienst wird also exportiert und steht Such- und Lernalgorithmen zur Verfügung, oder der Dienst wird nur von anderen Komponenten des Spielbretts benutzt, steht also nur intern zur Verfügung.

Durch einen Aufbau aus einzelnen Komponenten blieben diese austauschbar und die Spielbrett-Repräsentation kann einfach und vielseitig erweitert werden.

4.1.1 Zustands-Verwaltung

Um viele Samples der Ursprungssituation zu simulieren, muss dieser Zustand gespeichert werden und nach der Auswertung der Endposition wiederhergestellt werden. Zusätzlich muss die Ursprungssituation zwischen einzelnen Threads ausgetauscht werden.

Dazu dient die Schnittstelle Statefull (Quelltext-Ausschnitt 6), die es erlaubt, eine komprimierte Version der Daten eines Objektes zu speichern, zu ändern oder zu laden.

Ausschnitt 6 Schnittstelle zur Zustandsverwaltung

```
public interface Statefull {
  /**
   * save the current state of the object
   * @return Object that holds the state
   */
  Object saveState();

  /**
   * save the current state into an old saved state.
   * if the old state can't be reused,
   * the function may return a new object.
   * @param previous saved state
   * @return updated state or newly created state
   */
  Object updateSaved(Object saved);

  /**
   * restores the objects state to a saved one
   * @param saved state
   */
  void restoreState(Object saved);
}
```

Da viele der Dienste intern Daten verwalten, müssen auch diese gespeichert und wiederhergestellt werden: Über eine Komposition kann das Spielbrett-Objekt den Zustand all seiner Komponenten verwalten, d.h. alle Komponenten, die einen internen Zustand besitzen, müssen auch die Statefull-Schnittstelle implementieren, sodass das Spielbrett-Objekt deren Zustand in einer Liste speichern, die Einträge ändern und einzeln aus der Liste laden kann.

4.1.2 Dynamische Code-Erzeugung

Die lose Kopplung der Komponenten ist zum Testen verschiedener Erweiterungen wichtig. Allerdings muss auch die Ausführung der Aktionen auf dem Spielbrett höchst effizient sein.

Eine einfache Observer-Architektur, um Komponenten über Aktionen auf dem Spielbrett zu informieren, erzeugt sehr viel Overhead: der dynamische Aufruf einer Methode eines Subscribers steht oft in keinem Verhältnis zu dem eigentlichen Inhalt der Methode, da diese oft nur wenige Operationen ausführt, wie z.B. einzelne bit-Operationen oder einfach Tests, die im Fehlerfall keine weiteren Aktionen bewirken. Desweitern bleibt die Zusammensetzung der Komponenten nach der Konstruktion für den Verlauf des gesamten Go-Spiels fest.

Wird der Code, der die Komponenten zusammenfügt, zur Laufzeit erzeugt, kann sowohl die Austauschbarkeit der Komponenten, als auch eine schnelle Ausführung erreicht werden. Nach der Auswahl der Komponenten kann für jede Aktion auf dem Spielbrett und die Zustandsverwaltung der Code erzeugt werden, der die einzelnen Handler-Methoden in den Komponenten aufruft. Da dieser Code für die Laufzeit des Programmes statisch ist, kann dieser durch die Java VM besser optimiert werden und erreicht gleiche Geschwindigkeiten wie fest gekoppelte Komponenten.

4.1.3 Beispiel einer Komponente

Deklaration eines Dienstes

Die @Realize-Annotation gib an, dass der Dienst Funktionalitäten in eine Schnittstelle exportiert, mit der @Expose-Annotation kann angegeben werden, welche Methoden zu dieser Schnittstelle hinzugefügt werden.

Ausschnitt 7 Beispiel: Dienst-Deklaration

```
@Realize(Hashing.class)
public interface HashService extends Hashing {
  @Expose long hashValue();
  float getCollisionPropability();
}
```

Die Methode hashValue trägt also zur Realisierung der Hashing-Schnittstelle bei und kann von der Baumsuche verwendet werden, getCollisionPropability kann dagegen nur von anderen Komponenten benutzt werden, die eine Abhängigkeit gegenüber dem HashService angeben.

Durch die Angabe der Annotationen können die Schnittstellen, die das zusammengesetzte Objekt implementiert, bestimmt werden.

Deklaration einer Komponente

Zur Erzeugung der Komponenten wird das Dependency-Injection Framework google-guice[1] benutzt. Abhängigkeiten einer Komponente können auf diese Wei-

[1] google-guice – Dependency-Injection Framework (`http://code.google.com/p/google-guice/`)

se als Parameter eines Konstruktors angegeben werden, der mit @Inject annotiert ist.

Auch der Geltungsbereich einer Instanz wird von google-guice verwaltet. Die Annotation @GobanScoped gibt an, dass nur eine Instanz pro Spielbrett erzeugt wird, und nicht etwa eine neue Instanz für jede einzelne Komponente die eine Abhängigkeit zu dieser Komponente hat.

Ausschnitt 8 Beispiel: Dienst-Implementation

```
@GobanScoped
public final class ZobristHash
implements HashService, Statefull {
  private final IndexService idx;
  private final long[][] hashValues;
  private long hash = 0;

  @Inject
  ZobristHash(ArrayService ary, IndexService idx) {
    this.idx = idx;
    this.hashValues = getHashValues(ary);
  }

  // Implementation der Dienst-Schnittstelle
  public long hashValue() {
    return hash;
  }
```

Ereignis-Annotationen

Über verschiedene Annotationen wie @AfterPutStone, @OnRemoveStone oder @OnPass werden Methoden deklariert, die zu entsprechenden Ereignissen aufgerufen werden. So wird z.B. in dem Quelltext-Ausschnitt 9 toggleStone aufgerufen, wenn ein Stein auf das Brett gesetzt wurde, oder ein Stein entfernt wurde.

Unter Verwendung des Abhängigkeitsgraphen, den goolge-guice zur Verfügung stellt, und Laufzeit-Reflexion kann für jede Annotation „Glue-Code" generiert werden, der entsprechende Methoden auf den verschiedenen Objekten aufruft.

Ausschnitt 9 Beispiel: Ereignis-Deklaration

```
  @AfterPutStone
  @OnRemoveStone
  public void toggleStone(int id, int value) {
    hash = hash ^ hashValues[value - 1][id];
  }

  @AfterPutStone
  @OnPass
  public void togglePlayer() {
    hash = hash ^ 0x1;
  }

  @OnUpdateKo
  public void koHash(int id, int old) {
    if(old != 0) {
      hash = hash ^ hashValues[0][old];
      hash = hash ^ hashValues[1][old];
    }
    if(id != 0) {
      hash = hash ^ hashValues[0][id];
      hash = hash ^ hashValues[1][id];
    }
  }
```

Zustandsverwaltung

Die folgende Methoden sind Teil der Statefull-Schnittstelle und werden zur Verwaltung des Zustandes der Komponente aufgerufen. Wie auch für die annotierten Methoden wird für alle Komponenten, die Statefull implementieren, „Glue-Code" erzeugt, der den Zustand aller Komponenten verwaltet.

Ausschnitt 10 Beispiel: Zustandsverwaltung

```
  public long[] saveState() {
    return new long[]{hash};
  }

  public long[] updateSaved(Object saved) {
    assert saved instanceof long;
    long[] ls = (long[])saved;
    ls[0] = hash;
    return ls;
  }

  public void restoreState(Object saved) {
    assert saved instanceof long[];
    hash = ((long[])saved)[0];
  }
}
```

4.1.4 Konfiguration und Benutzung von Komponenten

Nachdem der Dienst deklariert wurde, muss dieser registriert und konfiguriert werden. Dazu stellt google-guice sogenannte Module-Deklerationen zu Verfügung. Dort werden Schnittstellen mit bestimmten Implementationen verknüpft (bind). Zusätzlich können bereits definierte Modules zu einem Module hinzugefügt werden (install).

Nun können über ein Injector-Objekt mit diesen Deklarationen neue Instanzen der Schnittstellen erzeugt werden. Dabei werden für die Abhängigkeiten der Klassen automatisch Objekte erzeugt und diese dem Konstruktor als Argumente übergeben. In Quelltextausschnitt 11 werden drei Module konfiguriert, um nun Objekte der HashingGoban-Schnittstelle erzeugen zu können.

Ausschnitt 11 Guice Module-Definition

```
public class BasicGobanModule extends AbstractModule {
    protected void configure() {
        install(GobanGeneratingModule.get("Basic", Goban.class));
        bind(ArrayService.class).to(BasicArray.class);
        bind(IndexService.class).to(BasicIndex.class);
        bind(ColorService.class).to(BasicColor.class);
        bind(BoardService.class).to(BasicBoard.class);
        bind(InfoService.class).to(BasicInfo.class);
        bind(ReaderService.class).to(BasicReader.class);
        bind(MoveGeneratorService.class).to(BasicGenerator.class);
        bind(ExecutorService.class).to(BasicExecutor.class);
    }
}
public class ZobristHashModule extends AbstractModule {
    protected void configure() {
        install(
            GobanGeneratingModule.get("Zobrist", HashingGoban.class));
        bind(HashService.class)
            .to(ZobristHash.class);
    }
}
public class GobanModule extends AbstractModule {
    protected void configure() {
        install(new BasicGobanModule());
        install(new ScoreModule());
        new DynamicValue<Integer>("size", 9).bindTo(binder());
        new DynamicValue<Float>("komi", 6.5f).bindTo(binder());
    }
}

final ConfigurationModule mod = new ConfigurationModule()
        .use(new GobanModule())
        .with(new ZobristHashModule());
Injector inj = Guice.createInjector(mod);
mod.set("komi", 0.5f);
mod.parse(args);
Goban goban = inj.getInstance(HashingGoban.class);
```

Mit GobanGeneratingModule.get(,,<Name>'', <Interface>.**class**) wird dabei ein Modul erzeugt, dass die in den vorherigen Absätzen beschrieben Abhängigkeiten und Annotationen benutzt, um daraus die zusammengesetzte Spielbrettklasse zu erzeugen. Die Klasse trägt den Namen <Name>Goban und implementiert die Schnittstelle <Interface>.

Mit Hilfe von DynamicValues und dem ConfigurationModule können den Komponenten zusätzlich noch Parameter übergeben werden und diese verändert werden.

4.1.5 Implementation

Mit Hilfe des Guice-Frameworks können über Reflexion die Abhängigkeiten der einzelnen Dienste gesammelt werden. Für die sich daraus ergebenden Dienst-Schnittstellen kann eine Liste der Funktionen, die die erzeugte Spielbrett-Klasse exportieren muss, erstellt und die Komponenten, die diese Funktionen implementieren, bestimmt werden. Die Methoden dieser Komponenten werden entsprechend ihren Annotationen zu einer Liste für das jeweilige Ereignis hinzugefügt. Zusätzliche wird eine Liste zur Zustandsverwaltung der Komponenten erzeugt.

Mit Hilfe des ASM-Frameworks[2], einem Java-Byte-Code-Generator, können diese Listen in Byte-Code für entsprechende Funktionsaufrufe umgewandelt werden. Die neu generierten Funktionen werden zu dem Spielbrett-Objekt zusammengefasst, das alle Anfragen an entsprechende Komponenten weiterleitet. Da diese Komponenten und Funktionsaufrufe sich nicht mehr ändern können, kann der erzeugte Code gut optimiert werden.

4.2 Details zu den wichtigsten Komponenten

4.2.1 Hilfskomponenten

Die in diesem Absatz beschriebenen Komponenten stellen selbst keine Funktionalität nach außen zur Verfügung, sondern übernehmen Dienste, die bei der Implementation anderer Komponenten gebraucht werden.

ArrayService

So wird für viele Komponenten ein Abbildung des Spielbrettes auf ein lineares Array benötigt. Diese Komponente hilft bei der Erzeugung und Verwaltung eines solchen Arrays.

Bei Go stellt es sich als sinnvoll heraus, nicht nur das Spielbrett in den Arrays zu speichern, sondern auch noch den Rand. Dieser kann oft wie eine Steinkette einer dritten Farbe (INVALID), die unendlich viele Freiheiten hat, behandelt werden. Dadurch vereinfachen sich viele Operationen und Algorithmen auf dem Spielbrett, da keine Sonderfälle für Steine am Rand behandelt werden müssen. Abbildung 4.2 zeigt die Anordnung des Arrays auf dem Spielfeld.

[2] ASM - Java Bytecode Manipulation Framework (`http://asm.ow2.org/`)

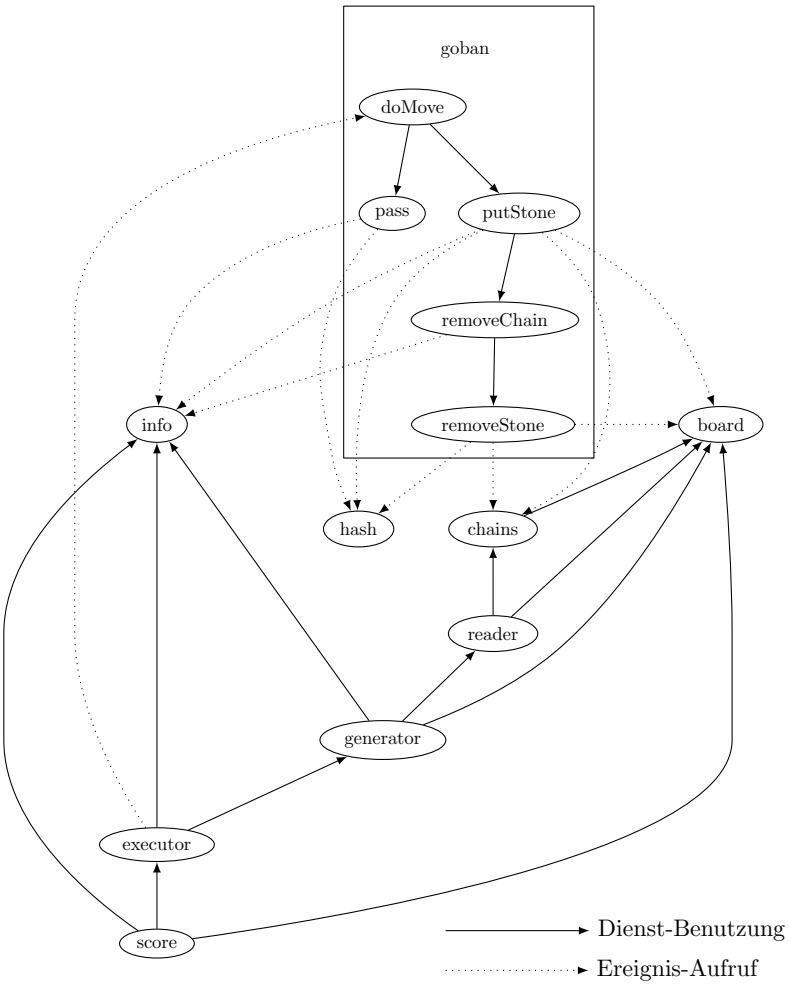

Abbildung 4.1: Interaktion der Komponenten

ColorService

Dieser Dienst bietet einfache Operationen auf den vier möglichen „Feld-Farben" (BLACK, WHITE, EMPTY, INVALID) an. Vergleich zweier Farben oder das Umkehren einer Farbe sind typische Dienste dieser Komponente.

IndexService

Der IndexService verwaltet Arrays, die verschiedene Indizes in das Spielbrett-Array darstellen, wie z.B. Indizes auf alle Rand-Felder oder alle Felder innerhalb des Spielbretts oder auch ein Array, das den Übergang zu den Feldern in allen vier Richtungen beinhaltet.

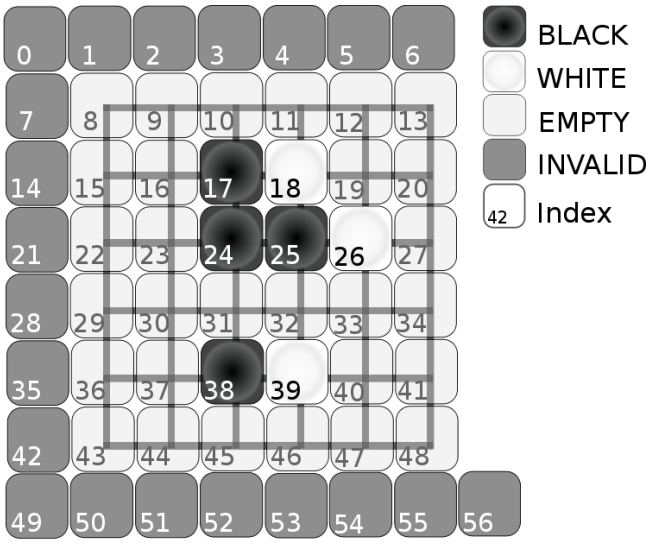

Abbildung 4.2: Anordnung eines Arrays auf dem Goban

ReaderService

Einfache taktische Informationen stehen über den ReaderService zur Verfügung. Informationen über Steinketten und deren Freiheiten können abgefragt werden. Dabei wurde eine Basisvariante implementiert, die Informationen stets neu berechnet, aber auch eine kompliziertere, verkettete Struktur (ChainReader), die sich Informationen über Steinketten merkt, allerdings mehr Speicher in Anspruch nimmt und beim Setzen eines Steines zur Aktualisierung der Strukturen mehr Rechenleistung benötigt.

Dennoch ist diese Steinketten-Struktur insgesamt etwas schneller als die Basisvariante, da zur Zuggeneration und Ausführung diese Informationen oft gebraucht werden. Zusätzlich stehen die gespeicherten Informationen für Expertenwissen und Mustererkennung zur Verfügung.

Marker

Mit Hilfe dieser Komponente können Felder im Spielbrett markiert werden. Abfragen und Setzen einzelner Markierungen sowie das Löschen aller Markierungen kann dabei in $O(1)$ durchgeführt werden. Dazu wird ein Integer-Array der Feldgrösse verwendet und ein Counter dient als Referenz für die Markierung. Wird ein Feld markiert, so wird der Feld-Position entsprechend der Wert einer Array-Zelle auf den Counter-Wert gesetzt. Die Abfrage nach einer Markierung erfolgt über einen Vergleich der Zelle mit dem Counter. Das Inkrementieren des Counters gleicht dem Löschen aller Markierungen.

Dies ist eine der wenigen Komponenten, von der nicht nur eine Instanz erzeugt wird, sondern für jede Komponente, die einen Marker benutzt, eine neue Instanz erzeugt wird.

4.2.2 Basiskomponenten

Diese Komponenten stellen die Basisfunktionen des Spielbretts zur Verfügung, die für das Ausführen eines Spiels notwendig sind.

BoardService

Speichert eine Repräsentation des Spielbretts, sodass die Farben der Spielfelder abgefragt werden können.

InfoService

Wird zur Verwaltung von weiteren Informationen wie nächster Spieler, Ko-Position oder Anzahl der gefangenen Steine verwendet.

ExecutorService

Führt einzelne Aktionen auf dem Spielbrett durch. Dazu wird der neue Stein gesetzt, es werden gegnerische Steine, die keine Freiheiten mehr haben, vom Spielbrett genommen und es findet ein Überprüfung nach einer Ko-Situation statt. Zusätzlich ruft der ExecutorService nach jeder Aktion den dem Ereignis entsprechenden ,,Glue-Code" auf, sodass andere Komponenten ihre Datenstrukturen ebenfalls auf die neue Situation aktualisieren können.

4.2.3 Zobrist Hash

Der wichtigste Teil der Implementation des Zobrist-Hash (siehe 2.1.4) wurde bereits im Quelltext-Ausschnitt 8 gezeigt.

Ein Bit des Hashwertes wird benutzt, um den aktuellen Spieler anzuzeigen, zusätzlich zu den Stein-Positionen wird noch die Ko-Position codiert, indem der Bit-String beider Steinfarben an dieser Position gesetzt wird, sowie die Differenz der bisher gespielten Pass-Züge.

4.2.4 Zug-Generator und Monte-Carlo-Sampling

Eine der wichtigsten Aufgaben einer Spielbrett-Repräsentation ist die Zuggenerierung und Validation, damit die Suche vorausschauend mögliche Züge erkennt. Zusätzlich müssen für die Monte-Carlo-Simulation tausende zufällige Züge generiert werden.

Der Zug auf ein Feld ist erlaubt, wenn mindestens einer seiner Nachbar-Felder *a*) leer ist, *b*) von einer eigenen Steinkette besetzt ist und diese eine weitere Freiheit hat oder *c*) von einer gegnerischen Steinkette besetzt ist, die keine weitere Freiheit besitzt. Bei dem MoveGeneratorService kann angefragt werden, ob einzelne Züge legal sind und eine Liste aller legalen Züge kann generiert werden.

Für das Monte-Carlo-Sampling muss zusätzlich noch überprüft werden, ob ein Zug ,,sinnvoll" ist. So sollten z.B. eigene Augen nicht zugesetzt werden, da sonst keine Steingruppe stabil ist und das Spiel nicht terminiert. Ein einfaches Verfahren, das die Stabilität der Gruppen gewährleistet, ist die Überprüfung auf Augen, die nur aus einem Feld bestehen. Zusätzlich können schlechte Züge durch Expertenwissen ausgeschlossen werden.

4.2.5 Wertung

Zur Bewertung am Spielende müssen die Gebiete gezählt werden. Unterschiede bei den weit verbreiteten japanischen und chinesischen Regelsätze führen zu leicht veränderten Ergebnissen in bestimmten Situationen, sodass zwei verschiedene Komponenten zur Auszählung nötig sind, die je nach Regelwerk eingesetzt werden.

4.2.6 AMAF-Daten

Für die Erweiterung der Suche ist das Mitführen einer All-Moves-As-First-Statistik wichtig. Diese geschieht in der AmafBoard-Komponente, die in der Sampling-Phase in einem Array den Spieler speichert, der einen Spielzug auf ein gewisses Feld das erste Mal ausführt. Diese Daten werden von der Suche ausgewertet und in den Spielbaum integriert.

4.2.7 Mustererkennung

Weitere Komponenten implementierten Aufgaben der Mustererkennung und des Expertenwissens. Über deren exportierte Schnittstellen kann die Suche auf diese Daten zugreifen. Thorsten Reinhardt untersucht in seiner Diplomarbeit verschiedene Ansätze zur Mustererkennung im Go untersucht. Dadurch entstehen weitere Komponenten, die der Suche zusätzliche Informationen bereitstellen.

4.3 Fazit

Das Spielbrett kann leicht durch weitere Komponenten erweitert werden und ist damit sehr flexibel. Die bereits implementierten Komponenten stellen eine gute Basis für die Baumsuche und deren Erweiterungen bereit. Über Modul-Definitionen und Abhängigkeitsdeklarationen des Such-Algorithmus werden die nötigen Komponenten zur Laufzeit ausgewählt und zu einem Objekt zusammengesetzt, auf dem höchst effizient operiert werden kann.

Kapitel 5

Monte-Carlo-Baumsuche

Auf einem im Speicher gehaltenen Spielbaum werden in einer Schleife folgende Schritte ausgeführt:

- Selektion und Expansion
- Sampling und Auswertung
- Zurückführen des Ergebnisses

Die einzelnen Aufgaben, wie das Verwalten des Baumes, aber auch die drei Suche-Schritte, werden dabei von einzelnen Klassen implementiert, sodass sich verschiedene Implementationen leicht austauschen lassen.

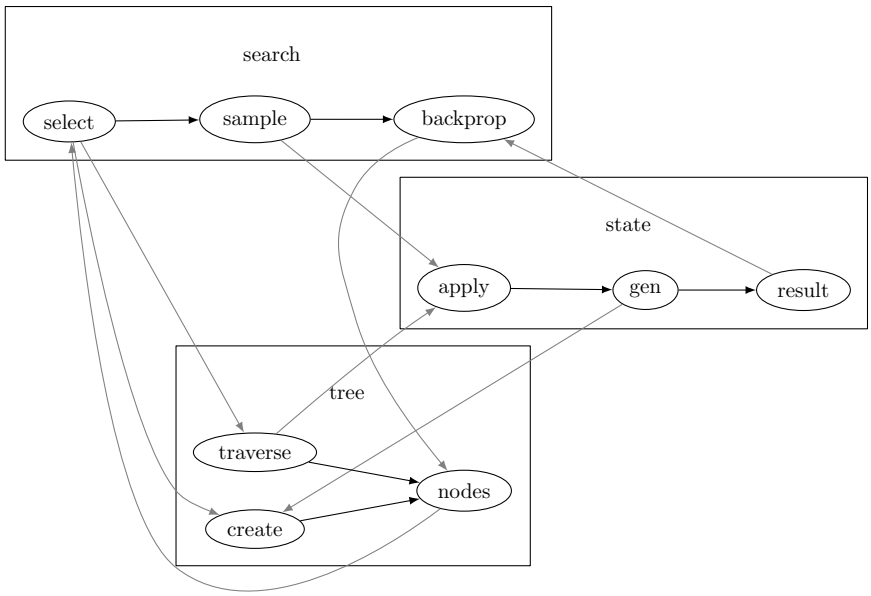

Abbildung 5.1: Interaktion zwischen Suche, Baum und Spielbrett

5.1 Ablauf

Ein Suchthread arbeitet auf einem lokalen Baum-Kontext, über den der Zugriff auf den Baum erfolgt. Der Kontext verwaltet eine Referenz auf den Baum und einen aktuellen Knoten sowie das zugehörige Spielbrett. Selektion und Expansion, Sampling, Auswertung und Rückführung des Ergebnisses werden durchgehend auf dem Baum ausgeführt. Quelltext-Ausschnitt 12 zeigt die Implementation des Suchablaufs.

Ausschnitt 12 Ausführung der Such-Schritte

```
public void run() {
  RootedContext<S, ? extends T> context
      = tree.createRootedContext(true);
  while (!Thread.interrupted()) {
    context.reset();
    run(context);
  }
}

public void run(RootedContext<S, ? extends T> context) {
  // select leaf node
  for (;;) {
    if (expand.isApplicable(context)) {
      expand.apply(context);
    }
    if (select.isApplicable(context)) {
      select.apply(context);
    } else {
      break;
    }
  }
  sample.apply(context);
  propagate.apply(context);
}
```

Der Baum wird zu jedem Zeitpunkt konsistent gehalten, sodass die Steuerung, nachdem die Suchzeit für einen Spielzug abgelaufen ist, den Zug mit der besten Statistik zurückgeben und die Wurzel im Baum verschieben kann, ohne die ausgeführte Suche zu unterbrechen, die nun auch in der Bedenkzeit des Gegners dem Baum Sampling-Ergebnisse hinzufügt.

5.2 Baumaufbau

Die Knoten enthalten Übergänge zu nachfolgenden Situationen und eine Referenz auf gelernte Bewertungen.

5.2.1 Baum-Kontext

Die Suche greift auf den gespeicherten Baum über ein Kontext-Objekt zu, dass neben dem Zeiger auf einen aktuellen Knoten im Baum auch ein Spielbrett

verwaltet. So muss nicht in jedem Baumknoten die Spielsituation gespeichert werden, es reicht aus wenn nur ein Spielbrett im Kontext gehalten wird. Bei jeder Traversierung im Baum, wird das Spielbrett entsprechend angepasst, sodass die Suche stets auf Übergänge, Knotenwerte und das dazugehörige Spielbrett zugreifen kann.

Dennoch ist über den Baumkontext die Suche von der Baumimplementation entkoppelt. So konnte z.B. einfach ein Baum mit Transpositionstabelle eingeführt werden, ohne die Suche zu verändern.

5.2.2 Knoten-Daten

Der Typ der Knoten-Daten wird über einen Typparameter angegeben und ist beliebig austauschbar und erweiterbar. Der Typ muss die Schnittstelle TreeData (siehe Ausschnitt 13) implementieren, um auf Ereignisse im Baumkontext zu reagieren.

Ausschnitt 13 Schnittstelle für Knoten-Daten

```
public interface TreeData<S extends State> {
  /** called when the value is first visited */
  void initialize(@Nullable TreeData<?> parent, int id, S state);

  /** called when the node is visited by a tree traversion */
  void visit(int id, S state);
}
```

Die visit-Funktion wird bei einer Traversierung im Baum, zu dem entsprechenden Knoten hin, aufgerufen. Wird ein neuer Knoten erstellt, so wird initialize aufgerufen. Somit können die Knoten-Daten aus einem Pool wiederverwendet werden.

5.2.3 Parallelisierung

Die Strukturen des Baumes und die gespeicherten Knoten-Daten sind dabei Thread-sicher implementiert, sodass mehrere Threads parallel auf dem Baum arbeiten können. Es wird also die Baum-Parallelisierung, wie im Abschnitt 2.4.1 vorgestellt, benutzt. Die in der Selektionsphase aufgerufene visit-Funktion erhöht den Besuch-Zähler des Knotens, sodass vermieden wird, dass mehrere Threads unnötigerweise gleichzeitig den selben Pfad testen.

Werden zwei Knoten gleichzeitig an einer Stelle in den Baum eingefügt, so geht einer der beiden verloren. In diesem unwahrscheinlichen Fall wird also etwas Rechenaufwand verschwendet, allerdings werden dadurch keine Locks gebraucht, die auch in allen anderen Fällen Ressourcen benötigen würden.

5.2.4 Transposition

Zur Transpositionserkennung wird eine Hashtabelle mit den 50.000 am häufigsten benutzten Knoten verwendet.

Die Knoten, die der gleichen Situation entsprechen, wurden wegen der unter 2.4.4 beschriebenen Problematik nicht selbst zusammengelegt, sondern nur Teile

ihrer Daten. Zusätzlich zu dem Durchschnittsergebnis $\bar{x}$ wurden auch die Kind-Knoten zusammengeführt. Dadurch tritt das Problem der fehlenden Exploration nicht auf, dennoch werden Teilbäume gemeinsam benutzt. Abbildung 5.2 zeigt ein Beispiel der zusammengeführten Kindknoten.

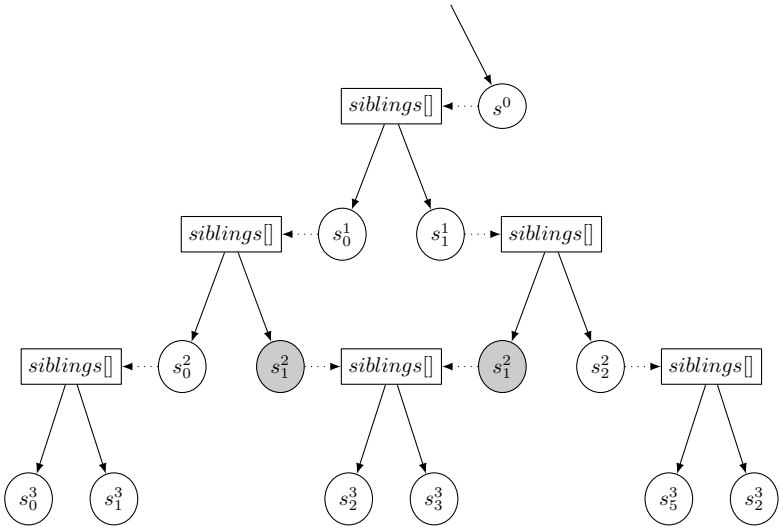

Abbildung 5.2: Zusammenführung der Knotenübergänge bei Transposition

5.3 Selektion und Expansion

Zur Auswahl des Pfades wird, wie unter Abschnitt 2.3.3 erläutert, jeweils der Kind-Knoten mit der höchsten oberen Konfidenz-Schranke gewählt, bis ein Blatt-Knoten erreicht wird.

Dieser wird expandiert, falls er schon einmal besucht wurde. Bei der Expansion werden die Übergänge lediglich vorbereitet. Erst wenn auf einen der Knoten zugegriffen wird, muss dieser erstellt werden.

5.3.1 Upper-Confidence-Bound

Das einfachste Auswahlverfahren nach UCT benutzt die Gewinn-Statistik des Knotens sowie die Anzahl der Besuche des Knoten selbst und des Elternknotens. Daraus kann die obere Konfidenz-Schranke berechnet werden (siehe Ausschnitt 14).

5.3.2 RAVE-gesteuerte Exploration

Neben der Mischung von AMAF-Statistik und UCT-Algorithmus, wie unter 2.4.2 beschrieben, wurde bei der Verwendung der AMAF-Statistik auch der UCB-Term weggelassen, sodass sich die Selektion nur aus AMAF-Daten und Gewinn-Statistiken zusammensetzt. Die Exploration erfolgt dabei über die AMAF-Daten, da automatisch Daten für verschiedene Züge gesammelt werden. Quelltext-Ausschnitt 15 zeigt die Verwendung der kombinierten Daten.

Ausschnitt 14 Berechnung des Upper-Confidence-Bound

```
class UCBFormula implements Formula {
  private final float ucbk;

  @Inject UCBFormula(
        @Config("ucbk")
        @Description("exploration factor for ucb formula")
        @Condition(CheckUCBK.class)
        @Default(f = 0.6f)
        float ucbk) {
    this.ucbk = ucbk;
  }

  public float apply(float rate, int visits, int parentVisits) {
    if(visits == 0) {
      return Float.POSITIVE_INFINITY;
    }
    return rate + ucbk * Math.sqrt(Math.log(parent)/visits);
  }
}
```

Um die Daten einzelner Zufallsspiele nicht überzubewerten, findet bei der Exploration ohne UCB-Term ein Ausgleich der Gewinn-Statistik statt. Dazu wird in der AvoidBadLuckFormula der Explorations-Term durch folgende Formel berechnet, wobei k_w und k_v geeignete Konstanten sind:

$$c_i = \frac{k_w + wins_i}{k_v + visits_i} \tag{5.1}$$

5.4 Sampling und Auswertung

In der zweiten Phase des Such-Algorithmus wird ein zufälliges Spiel ausgeführt und bewertet. Für die Zuggeneration und Bewertung der Endsituation werden entsprechende Funktionen der Spielbrett-Klasse aufgerufen.

5.4.1 Light vs. Heavy

Bei den Zufallsspielen unterscheidet man zwischen Light- und Heavy-Playout. Bei Light-Playouts wird die Priorität auf einfache Zugmanöver und maximale Geschwindigkeit gelegt. Die Auswahl der Züge erfolgt zufällig, es wird nur auf Legalität eines Zuges sowie das Bewahren von Augen geachtet.

Bei Heavy-Playouts wird dagegen eine kompliziertere Zugauswahl benutzt, was den Aufwand eines Zufallsspiels erhört. Anstatt Züge rein zufällig auszuführen, werden wichtige Züge, wie z.B. das Retten einer Steingruppe im Atari, priorisiert ausgeführt oder Muster verwendet um „bessere" Züge zu erzeugen. Dadurch ähnelt das Zufallsspiel mit Heavy-Playouts eher einem echten Go-Spiel, sodass die generierten Gewinn-Statistiken der Zufallsspiele aussagekräftiger sind.

Es wurde sowohl die leichtfüßige Variante implementiert, wie auch eine etwas schwerere, die taktisch wichtige Züge bevorzugt.

Ausschnitt 15 Auswahl nach RAVE

```
class ValuatedCombined implements Valuation<WinAMAFData> {
  private final float bias;
  private final Formula formula;

  public float apply(WinAMAFData parent, int id,
        WinAMAFData child) {
    int nodeCount, raveCount;
    int nodeVisits, raveVisits;
    float nodeRate, raveRate;
    float nK, rK, rate, visit, count;

    nodeCount = parent.getVisits();
    if (child != null) {
      nodeVisits = child.getVisits();
      nodeRate = child.getRating();
    }
    raveCount = parent.getAMAFCount();
    raveVisits = parent.getAMAFVisits(id);
    raveRate = parent.getAMAFRating(id);

    if (raveVisits == 0) {
      return Float.POSITIVE_INFINITY;
    } else {
      rK = raveVisits /
          (raveVisits+nodeVisits + bias*raveVisits*nodeVisits);
      nK = 1f - rK;
    }

    // weighten rave vs. normal data
    rate = nK * nodeRate + rK * raveRate;
    visit = nodeVisits + rK * raveVisits;
    count = nodeCount + rK * raveCount;
    return formula.apply(rate, (int)visit, (int)count);
  }
}
```

5.4.2 AMAF-Heuristik

Am Ende des Spiels wird neben dem Gewinner optional auch die All-Moves-As-First Belegung der Felder ausgewertet, die damit in die RAVE-Heuristik einfließen kann.

5.5 Zurückführen des Ergebnisses

Die dritte Phase der Suche aktualisiert den Baum mit den gewonnenen Informationen. Dabei werden in umgekehrter Reihenfolge, also mit dem Blatt beginnend, die ausgewählten Knoten besucht und aktualisiert.

Die Gewinn-Statistik kann für die Knoten, die von dem Gewinner verursacht wurden, einfach erhöht werden.

5.5.1 RAVE-Aktualisierung

Die Aktualisierung der RAVE-Felder ist etwas aufwendiger. Mit dem Blatt-Knoten beginnend, wird die AMAF-Belegung der Felder benutzt um die RAVE-Werte der Knoten zu aktualisieren. Beim Aufsteigen im Baum muss zusätzlich der entsprechende Zug zu der Belegung hinzugefügt werden.

Zur Aktualisierung der RAVE-Werte muss für jeden der Züge in der AMAF-Belegung, die der handelnden Farbe im Blatt-Knoten entspricht, die Gewinn- und Besuchsstatistik entsprechend angepasst werden. In Ausschnitt 16 wird der dafür relevante Quelltext gezeigt.

Ausschnitt 16 RAVE Aktualisierung

```
public class AMAFPropagation
implements Propagation<AMAFBoard, AMAFData> {
  public void apply(
        RootedContext<AMAFBoard,AMAFData> context) {
    AMAFResult amaf = context.getState().getAmafResult();
    List<AMAFData> vals = context.getValuePath();
    List<Integer> acts = context.getActionPath();
    for (int i = vals.size() - 1; i >= 0; i--) {
      AMAFData val = vals.get(i);
      val.getAMAFValue().add(val.getActor(), amaf);
      if(i > 0) {
        amaf.addMove(acts.get(i - 1),
                     vals.get(i - 1).getActor());
      }
    }
  }
}
public final class AMAFAggregate
implements Aggregate<AMAFResult> {
  private final AtomicIntegerArray wins;
  private final AtomicIntegerArray visits;
  private final AtomicInteger count;
  public void add(int actor, AMAFResult result) {
    boolean win = result.getWin() == actor;
    for (int i = 0; i < result.moves.length; i++) {
      if (result.moves[i] == actor) {
        count.incrementAndGet();
        visits.incrementAndGet(i);
        if (win) {
          wins.incrementAndGet(i);
        }
      }
    }
  }
}
```

5.6 Fazit

Die Implementation des grundlegenden UCT-Algorithmus wurde flexibel gestaltet und mit verschiedene Heuristiken zur Verbesserung des Ergebnisses und Steigerung der Effizienz erweitert.

Kapitel 6

Experimente und Ergebnisse

Um die KI zu bewerten wurden Spiele sowohl gegen andere Go-Computer als auch gegen menschliche Spieler durchgeführt.

6.1 Test mit GnuGo

Zum Testen wurden Spiele gegen GnuGo[1] durchgeführt. GnuGo ist ein Open-Source Programm, dass auf viel Expertenwissen und lokaler α-β-Suche aufbaut. Der Vergleich der Implementation mit einer klassisch aufgebauten Go-KI ist besonders interessant, da die Stärken und Schwächen der verschiedenen Ansätze gut zum Vorschein kommen.

GnuGo (Version 3.8 mit `--level 10`) besitzt eine Spielstärke von ca. 9 kyu. Mit dem UCT-Algorithmus ohne Erweiterungen, gewinnt die KI selbst bei länger Rechenzeit nur ca. ein Drittel der Spiele gegen GnuGo (siehe Tabelle 6.1). Mit Erweiterungen und optimierten Parametern spielen selbst bei einer kurzen Rundenzeit von unter 10s die beiden Programme gleich gut, obwohl kompliziertere Mustererkennung noch nicht realisiert wurde und nur wenig Expertenwissen in die KI eingeflossen ist.

Um unabhängig von der Rechnerleistung zu testen, wurden die Spiele jeweils mit gleicher Anzahl von Zufallsspielen ausgeführt. Für 50.000 Zufallsspiele werden, je nach Rechner, 5 bis 30 Sekunden benötigt. Wegen der hohen Speicherauslastung bei größeren Bäume nimmt die Anzahl der Spiel pro Sekunde bei höherer Sample-Zahl ab.

6.2 Erweiterungen

Durch verschiedene Erweiterungen und Heuristiken kann das Ergebnis des reinen UCT-Algorithmus deutlich verbessert werden, wie Tabelle 6.1a zeigt. Auf die Auswirkungen der einzelnen Erweiterungen wird in den folgenden Abschnitten eingegangen.

[1] GnuGo - GNU Go Computer (`http://www.gnu.org/software/gnugo/`)

#Samples	% Gewinne
nur einfacher UCT:	
10000	9 ± 3
50000	22 ± 3
200000	28 ± 4
500000	36 ± 5
mit Erweiterungen:	
50000	53 ± 3
200000	63 ± 8
1000000	78 ± 9

(a) variable Sample-Anzahl

ucbk	% Gewinne
0.3	8 ± 7
0.4	22 ± 3
0.5	23 ± 3
0.6	22 ± 3
0.8	20 ± 3
1.0	19 ± 3
1.3	16 ± 3
2.0	15 ± 2

(b) Einfluss des Explorations-Faktors

Tabelle 6.1: Spielstärke des UCT-Algorithmus

	Light	Heavy
nur UCT	22 ± 3	46 ± 3
mit RAVE	44 ± 3	50 ± 2

Tabelle 6.2: Light & Heavy, je 50.000 Samples

6.2.1 Light vs. Heavy

Die Verbesserung der Zugauswahl in der Sample-Phase bringt sowohl für den einfach UCT-Algorithmus als auch mit anderen Erweiterungen eine deutliche Verbesserung, obwohl dieser noch sehr einfach gehalten wurde (siehe Tabelle 6.2). Erweiterungen zur Anwendung von weiterem Expertenwissen und maschinellem Lernen in der Sampling-Phase könnten die Gewinn-Rate sicher noch weiter steigern.

6.2.2 RAVE

Tabelle 6.3 zeigt die Ergebnisse der Verwendung von RAVE mit Light-Playouts. Für die erste Spalte wurde die Exploration durch die Konfidenz-Schranke verwendet, für die zweite Spalte wurde nur die Exploration durch die gesammelten RAVE-Werte verwendet.

Das Ergebnis zeigt, dass bei geringer Anzahl von Samples die zusätzliche Exploration, die der UCB-Term mit sich bringt, hinderlich ist. Bei einer hohen Anzahl an Samples ist sie allerdings wichtig, damit die Konvergenz zur Hauptvariante sichergestellt wird.

#Samples	% Gewinne	
Exploration mit	UCB+RAVE	RAVE
5000	22 ± 4	27 ± 3
20000	33 ± 4	34 ± 3
50000	37 ± 4	44 ± 3
200000	51 ± 6	44 ± 5

Tabelle 6.3: Ergebnisse RAVE

#Threads	Zeit
1	$31s$
2	$16s$
3	$10s$
4	$7s$
8	$7s$

Tabelle 6.4: Parallelisierung – 200000 Samples, Quad-Core

Verwendete Transpostion	% Gewinne	
	50k	200k
direktes Zusammenführen	8 ± 5	11 ± 7
ohne Transposition	51 ± 2	56 ± 3
zusammengeführte Kindlisten	53 ± 3	63 ± 8

Tabelle 6.5: Ergebnisse Transposition

6.2.3 Parallelisierung

Die Verwendung der Baum-Parallelisierung führt nicht zu einer Steigerung der Spielstärke, nutzt aber vorhandene Ressourcen besser aus, sodass die Zeit, die die Suche beansprucht, zurückgeht. Tabelle 6.4 zeigt das Ergebnis auf einer Quad-Core Maschine. Der Overhead für die parallele Ausführung ist minimal, da keine Locks verwendet werden.

6.2.4 Transposition

Bei Anwendung des reinen UCT-Algorithmus traten bei 50.000 Simulationen zwischen 5.000 und 20.000 Transpositionen auf. Werden andere Heuristiken benutzt, um die Suche einzugrenzen, ist diese Zahl etwas niedriger.

Durch das Zusammenführen der Teilbäume wurde nur eine leichte Verbesserung des Ergebnisses erzielt (siehe Tabelle 6.5). In der Tabelle wird auch die Auswirkung des direkten Zusammenführens der Knoten und des damit verbundenen Explorations-Fehlers deutlich.

Um eine weitere Verbesserung zu erzielen, müssten zusätzlich noch alle Knoten oberhalb der Transposition von jedem zurückgeführten Ergebnis beeinflusst werden. Dies jedoch kann wiederum zu Situationen führen, die weitere Exploration verhindern.

6.3 Analyse gespielter Spiele

Bei Spielen gegen die KI oder der Analyse von Spielen gegen GnuGo, fallen die Stärken und Schwächen der KI auf:

- Stärke in gewinnentscheidenden Kämpfen

  Ist nur ein Kampf, der über Sieg und Niederlage entscheidet auf dem Brett und die restlichen Gruppen sind stabil, findet die KI oft sehr gute Zug-Kombinationen, die Schwächen des Gegners ausnutzen, die GnuGo oder

auch menschliche Spieler oft übersehen. Dies zeigt deutlich das Potential des UCT-Algorithmus.

- Intuitives Spiel

  Wo ein menschlicher Spieler seine Intuition benutzt, um global gute Züge zu finden und klassische Go-KIs oft Probleme haben, diese Züge zu entdecken, findet der UCT-Algorithmus oft auch diese „intuitiv" richtigen Züge, da diese die Auswirkungen auf die Wahrscheinlichkeiten des Gebietsgewinns in großen Regionen des Bretts beeinflussen.

  Zusätzlich wird die Sicherheit eines Gebietes gut modelliert, da auch in die Zufallsspiele die Sicherheit des Gebiets direkt mit einfließt. Klassische Evaluationen weisen hier größere Schwächen auf, da diese unsicheres Gebiet nur schlecht Einordnen können.

- Schwäche in Zwei-Fronten-Kämpfen

  Oft werden Züge ausgeführt, die zwar auf beide Fronten gleichzeitig wirken, aber bei keinem der Kämpfe einen entscheidenden Vorteil bringt, sodass oft beide Kämpfe verloren werden. Dies ist darauf zurückzuführen, dass nun in beiden Kämpfen die Wahrscheinlichkeit eines Sieges gestiegen ist und diese zusammengerechnet höher ist als für einen Zug, der nur in einem der Kämpfe die Wahrscheinlichkeit des Sieges erhöht, den anderen Kampf jedoch aufgibt. Es wäre oft von Vorteil, sich für eine Seite zu entscheiden. Einen Kampf mit einer großen Wahrscheinlichkeit zu gewinnen (z.B. $p_1 = 0.7$), den anderen aber sicher (z.B. $p_2 = 0.1$) zu verlieren, ist oft vorteilhaft, gegenüber zwei unsicheren Ergebnissen (z.B. $p_1, p_2 = 0.5$).

- Verzögerung der Entscheidungen

  Oft werden Züge gespielt, die eine Antwort erzwingen, nach der korrekten Antwort aber keine Verbesserung der Stellung ergeben, oft aber der KI Möglichkeiten im späterem Spiel nehmen. Auch hier wird die Wahrscheinlichkeit für einen Gewinn nur scheinbar gesteigert, indem damit gerechnet wird, dass der Gegner einen Fehler begehen könnte.

- Lange, einfach Zugfolgen

  Im Go existiert eine Reihe von Situationen (semeai, Leiter), in denen eine lange Zugfolge notwendig ist, um ein Ergebnis herbeizuführen. Diese Zugfolgen sind einfach zu entdecken, da sie gewissen Mustern folgen und keinen hohen Verzweigungsgrad aufweisen.

  Dennoch versagt der UCT-Algorithmus hier, da er bei jedem Zug der Zugfolge erst alle anderen Züge als schlecht evaluieren muss, bevor er den nächsten Zug der Folge testen kann. Weiteres Experten-Wissen müsste für solche Situationen eingeführt werden, allerdings wäre ein solcher Ansatz sehr statisch und aufwendig, um alle Spezialfälle abzudecken.

6.4 Fazit

Der UCT-Algorithmus kann ohne viel Expertenwissen gute Ergebnisse erzielen und mit Hilfe der verschiedenen Heuristiken kann die KI auch klassische KIs

mit viel Expertenwissen schlagen. Mustererkennung und Expertenwissen können aber auch hier das Ergebnis zusätzlich steigern.

Dennoch treten einige Schwächen im Spiel auf, für deren Überwindung neue Ansätze mit in den Algorithmus einfließen müssen.

Kapitel 7

Zusammenfassung und Zukünftige Beiträge

Dieser Abschnitt gibt noch einmal einen Überblick über erreichte Ziele, es wird aber auch auf die Grenzen des Ansatzes eingegangen, sowie das Potential für zukünftige Verbesserungen aufzeigt.

7.1 Überblick meiner Beiträge

Allgemeines

- Grundlage für weitere Arbeiten die auf der Monte-Carlo-Baumsuche aufbauen
- Implementation und Evaluation verschiedener Erweiterungen und Heuristiken
- Weiterführung der Transpositions-Idee für die Baumsuche

Infrastruktur

- Projektseite (`https://dev.spline.de/trac/kifoo/wiki`)
- Build- und Source-Control-Management
- Automatisiertes, verteiltes Testen

Programmierung

- Ablaufsteuerung
- Spielbrett-Komponenten
- Monte-Carlo-Baumsuche
  - Parallelisierung der Suche
  - Transpositionsverfahren
  - verschiedene Ansätze zur Auswertung der Samples
  - verschiedene Heuristiken, um die Suche zu leiten

7.2 Probleme

Durch die im Abschnitt 6.3 beschriebene Analyse gespielter Spiele, konnten die Schwächen der KI ausfindig gemacht werden. Auch wenn weiteres Expertenwissen spezielle Schwächen teilweise einschränken könnte, sollten dennoch die grundlegenden Probleme der Suche gelöst werden.

7.2.1 Horizont-Effekt

Wichtige Entscheidungen werden oft über den Such-Horizont hinaus verschoben (siehe 6.3). Dies tritt sowohl bei langen Zugfolgen wie auch beim Kampf an mehreren Teilen des Spielbretts auf. Zwar gibt es nicht wie bei der Evaluation in der α-β-Suche einen scharfen Horizont, dennoch können Ergebnisse in dem Suchbaum nach hinten geschoben werden, sodass deren Evaluation ungenau wird.

7.2.2 Lokale Ergebnisse

Die Unterteilung in lokale und globale Auswirkungen eines Spielzuges ist im Go problematisch, dennoch können lokale Berechnungen auch im globalen Kontext ihre Gültigkeit bewahren.

Ein großer Vorteil des UCT-Algorithmus ist es, dass global wichtige Züge und deren Auswirkungen auf lokale Kämpfe gut erkannt werden. Allerdings wird mit der Neuberechnung der lokalen Ergebnisse an vielen verschiedenen Stellen des Baumes Rechenzeit verschwendet und das Auftreten des Horizont-Effekts unterstützt.

7.3 Zukünftige Projekte

Der UCT-Algorithmus stellt ein neues Werkzeug dar, das für das Go-Spiel gut geeignet scheint. Die Verbindung mit anderen Methoden, sowie die Anwendung auf verschiedene Probleme muss noch genauer untersucht werden, damit der UCT-Algorithmus so ausreift, wie es die Minimax-basierenden Such-Algorithmen heute sind.

7.3.1 Mustererkennung

Schon einfaches Expertenwissen in den Zufallsspielen hat die Spielstärke deutlich gesteigert. Die Anwendung von Mustererkennung und maschinellem Lernen zur Verbesserung der Samples sowie für die Einschränkung des Suchraums könnte die Stärke der KI weiter vorantreiben. Maschinelles Lernen im Kontext des Monte-Carlo-Algorithmus stellt dabei eine interessante Herausforderung dar.

Im Rahmen seiner Diplomarbeit legt Thorsten Reinhard die Grundsteine für die Mustererkennung und deren Anwendung im UCT-Algorithmus, Potential für weitere Untersuchungen und Verbesserungen ist hier weiterhin gegeben.

7.3.2 Zug-Datenbanken

Durch die Anwendung gelernter Zugabfolgen in der Eröffnung (siehe Fuseki und Joseki, 1.2.2) oder durch lokal geschickte Zugfolgen (Tesuji) können viele

Situationen vereinfacht werden.

Auch der UCT-Algorithmus könnte von der Erhebung einer Zug-Datenbank und deren Anwendung in der Suche profitieren. Das Wissen der Datenbank nicht nur statisch anzuwenden, sondern auf die Situation anzupassen, ist dabei im Go-Spiel von besonderer Bedeutung. Die Wichtigkeit, die Abfolgen anzupassen, wird durch den Spruch verdeutlicht, man solle nicht Josekis lernen, sonder **durch** diese lernen.

7.3.3 Lokalität

Eine Möglichkeit, lokale Ergebnisse zu berechnen und deren Gültigkeit im globalen Kontext zu bestimmen, könnte den nächsten Durchbruch des UCT-Algorithmus darstellen. Da aus den gesammelten Zufallsspielen auch Informationen über die Stabilität von Gruppen und deren Einfluss extrahiert werden könnte und damit Rückschlüsse auf den globalen Einfluss von Zügen gezogen werden können, scheint der Monte-Carlo-Ansatz sehr geeignet, um das Problem zu lösen.

Eine Beschreibung und Codierung der Größen wie „Stabilität", „Einfluss" und „Lokalität" müssten gefunden werden, Ansätze für deren Erhebung aus den Zufallsspielen müssten untersucht und deren Folgerungen in den Aufbau des Suchbaums integriert werden.

7.3.4 Weitere Anwendungen

Es gibt eine Vielzahl an Zweispieler-Nullsummen-Spielen, bei denen klassische Spiele-KI-Methoden versagen, wie z.B. die Spiele Arimaa oder Hex. Auch hier könnte der UCT-Algorithmus angewandt und einige der Heuristiken übertragen werden. Aber auch für Mehrspieler- und zufallsbasierte Spiele oder bei der Programmierung einer allgemeinen Spiele-KI basierend auf WDL [14] ist UCT durch seine schwache Annahme eines Markowschen Entscheidungsproblems interessant.

7.4 Schlusswort

Die Analyse des UCT-Algorithmus und der einzelnen Heuristiken zeigt die Stärken und Schwächen der Ansätze auf. Dies stellt aber erst den Anfang eines Zyklus dar, jetzt können die gefundenen Schwächen überwunden und die Stärken ausgebaut werden.

go, kifoo! – and find your style

– Daniel Wäber

Literaturverzeichnis

[1] Markov decision process. http://en.wikipedia.org/wiki/Markov_decision_process

[2] INRIA, France: Modification of UCT with Patterns in Monte-Carlo Go. Version: November 2006. http://hal.inria.fr/docs/00/12/15/16/PDF/RR-6062.pdf. 2006 (6062). – Forschungsbericht

[3] Abramson, B.: Expected-Outcome: A General Model of Static Evaluation. In: IEEE Transactions on Pattern Analysis and Machine Intelligence 12 (1990), Nr. 2

[4] Allis, Louis V.: Searching for Solutions in Games and Artifcial Intelligence. University of Limburg, 1994 http://fragrieu.free.fr/SearchingForSolutions.pdf

[5] Bouzy, B. ; Helmstetter, B.: Monte-Carlo Go Developments, Kluwer Academic, 2003

[6] Bouzy, Bruno: Associating Domain-Dependent Knowledge and Monte Carlo Approaches within a Go Program. In: In: Joint Conference on Information Sciences, 2003, 505–508

[7] Brügmann, Bernd: Gobble. Descirption on Computer Go Ladder. http://www.cgl.ucsf.edu/home/pett/go/Programs/Gobble.html

[8] Brügmann, Bernd: Monte Carlo Go. ftp://ftp.cgl.ucsf.edu/pub/pett/go/ladder/mcgo.ps. Version: 1993

[9] Chaslot, Guillaume M. ; Winands, Mark H. ; Herik, Jaap H. d.: Parallel Monte-Carlo Tree Search. In: Proceedings of the 6th International Conference on Computer and Games, Springer, 2008

[10] Childs, Benjamin E. ; Brodeur, James H. ; Kocsis, Levente: Transpositions and Move Groups in Monte Carlo Tree Search. In: Hingston, Philip (Hrsg.) ; Barone, Luigi (Hrsg.) ; IEEE (Veranst.): IEEE Symposium on Computational Intelligence and Games IEEE, 2008. – ISBN 978–1–4244–2974–5, 389–395

[11] Garlock, Chris: COMPUTER BEATS PRO AT U.S. GO CONGRESS. In: American Go Association News (2008). http://www.usgo.org/index.php?%23_id=4602

[12] Gelly, Sylvain ; Silver, David: Combining Online and Offline Knowledge in UCT. In: In Zoubin Ghahramani, editor, Proceedings of the International Conference of Machine Learning, 2007, S. 273–280

[13] Kaelbling, L. P. ; Littman, M. L. ; Moore, A. W.: Reinforcement Learning: A Survey. http://www.cs.washington.edu/research/jair/volume4/kaelbling96a-html/node6.html. Version: May 1996

[14] Kulick, Johannes: Erweiterung von GDL und Lösungsstrategien für unbekannte Spiele. http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/Betreute_Arbeiten/Bachelorarbeit-Kulick.pdf. Version: 2009

[15] Levente Kocsis, Csaba S.: Bandit based Monte-Carlo Planning, Springer, 2006

[16] Werf, Erik van d.: AI techniques for the game of Go. Universitaire Pers Maastricht, 2004 http://erikvanderwerf.tengen.nl/pubdown/thesis_erikvanderwerf.pdf

[17] Zobrist, A. L.: A New Hashing Method with Application for Game Playing / University of Wisconsin. Version: 1970. http://www.cs.wisc.edu/techreports/1970/TR88.pdf. 1970 (88). – Forschungsbericht

Quelltext-Ausschnitte und Algorithmenverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis