TECHNISCHE
UNIVERSITÄT
DARMSTADT

Bachelor-Arbeit

Automatische Generierung von Zugpatterns aus Spieldatenbanken für Go

von
Michael Wächter

Betreuer & Prüfer:
Prof. Dr. Johannes Fürnkranz

15. Oktober 2008

Erklärung

Hiermit versichere ich, die vorliegende Bachelor-Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus 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

1. Einleitung

> „Bevor jemand anfängt Go zu studieren, sieht er nur schwarze und weiße Steine auf einem Stückchen Holz. Nachdem man angefangen hat Go zu studieren, sieht man mehr: Tesuji, Joseki, gute Form, schlechte Form und tausend andere Dinge. Wenn man dann wahre Meisterschaft erlangt hat, sieht man nur schwarze und weiße Steine auf einem Stückchen Holz.“ - Unbekannter Autor

Ziel dieser Bachelorarbeit ist die Erstellung eines Algorithmus, der zu einer gegebenen Stellung des Brettspiels Go eine Menge von in dieser Stellung geeigneten Zügen ausgibt. Es ist nicht notwendig, dass der Algorithmus den besten Zug findet, sondern lediglich dass er eine Menge an Zügen ausgibt, in der sich mit einer möglichst hohen Wahrscheinlichkeit der beste Zug befindet.

Dieses Ziel soll durch die Extraktion von Wissen aus Datenbanken von Go-Partien erreicht werden. Das extrahierte Wissen wird in Form von Zugpatterns repräsentiert.
Die Arbeit behandelt das Aussehen der Patterns, ihre Extraktion aus der Datenbank, die Technik, mit der sie mit einer gegebenen Spielsituation gematcht werden können, die Bewertung der Güte der Patterns und die Vorhersage eines Zuges. Weiterhin wird auf die Möglichkeiten der Anwendung und der Erweiterung des Algorithmus eingegangen.
Go wurde deshalb gewählt, weil Computer-Go, obwohl weltweit bereits sehr viel Forschungs- und Programmierarbeit geleistet wurde ([19, Abschnitt 1. Introduction]), immer noch ein recht offenes Feld der KI-Forschung ist. Ein wichtiger Bestandteil menschlichen Spielens ist die Mustererkennung ([19, Abschnitt 2.4.3. Quality and quantity of human knowledge]) und daher liegt es nahe die Verwendung von Patterns für Go-KIs zu untersuchen.

Abschnitt 1 geht auf Computer Go, seine Geschichte und den Nutzen, den Verfahren des Maschinellen Lernens auf diesem Gebiet haben können, ein. Außerdem werden ein paar Begriffe, die in dieser Arbeit von Bedeutung sind, erklärt.
Abschnitt 2 stellt das Ziel dieser Arbeit vor und gibt einen kleinen Überblick, welche Ansätze mit selber oder ähnlicher Zielstellung bisher existieren.
Abschnitt 3 erklärt, wie der im Rahmen dieser Arbeit implementierte Algorithmus funktioniert.
Abschnitt 4 geht auf die Performanz des Algorithmus, d.h. die Qualität, mit der er seine Aufgabe verrichtet, sowie sein Zeit- und Speicherverhalten, ein.
Abschnitt 5 listet schließlich auf, welche Möglichkeiten zur Verbesserung der Performanz des Algorithmus noch bestehen und wie der Algorithmus in einem kompletten Go-KI-

Algorithmus eingesetzt werden kann.
Für Leser, die mit den Regeln von Go nicht vertraut sind, werden diese in Anhang B erklärt. Sie sind für das Verständnis dieser Arbeit eventuell hilfreich.

1.1. Warum Go?

Sowohl Go als auch Schach sind schon seit längerer Zeit Untersuchungsgegenstand im Gebiet der Künstlichen Intelligenz. Computer-Go wird bereits seit den 1960er Jahren untersucht ([5, Abschnitt 3.1. History of Computer Go]). Dennoch entzieht es sich bis heute einem so weitreichenden Zugang wie Schach.

Im Computerschach konnte man bisher große Erfolge verbuchen. Zu Beginn der Erforschung dieses Gebiets fanden zunächst Debatten statt, ob eine sog. A-Strategie oder eine B-Strategie zum Erfolg führen würde. Erstere bezeichnet das vergleichsweise unintelligente Brute-Force-Durchsuchen des Spielbaums, letztere konzentriert sich darauf plausible Züge in Erwägung zu ziehen und der menschlichen Weise des Denkens und Schlussfolgerns nahezukommen.
Versuche mit reinen B-Strategien stellten sich als schwierig umzusetzen und wenig erfolgreich heraus ([21, Abschnitt Search versus Knowledge]). A-Strategien erwiesen sich als erfolgreicher. Die von vielen aktuellen Schach-KIs verwendeten Algorithmen (bspw. der Alphabeta-Algorithmus, Killer-Heuristik und Null-Move-Heuristik) sind im Kern suchintensiv und nicht auf Schach spezialisiert. Schachwissen ist nur zur Bewertung von Blättern des Alphabeta-Algorithmus unentbehrlich. 1997 schaffte es der erste Computer, Deep Blue, den amtierenden Weltmeister in einem Match unter Turnierbedingungen zu schlagen. Deep Blue verwendete dazu Spezialhardware und berechnete über 100 Mio. Stellungen pro Sekunde ([7]).

Im Computer-Go war man vergleichsweise unerfolgreich. Der Transfer der im Schach erfolgreichen Algorithmen bzw. reine A-Strategien führten hier nicht zum Erfolg. Hierfür wird eine Reihe von Gründen angeführt (vgl. [26, Abschnitt Obstacles to high level performance]). Das Brett ist wesentlich größer als im Schach und im Schnitt stehen bedeutend mehr Zugalternativen zur Wahl. Sie sind zwar nur zu einem kleinen Teil sinnvoll, aber es ist nicht möglich einen großen Teil von ihnen effizient auszuschließen. Der Verzweigungsfaktor des Spielbaums ist daher sehr groß und lässt ihn schon bei sehr kleinen Suchtiefen auf eine nicht beherrschbare Größe anwachsen.
Hinzu kommt, dass eine verhältnismäßig kleine Suchtiefe wie im Schach (z.B. 12 Halbzüge) im Go nicht ausreicht. In den meisten Stellungen ist es einem guten menschlichen Spieler möglich die weitere Entwicklung des Spiels über wesentlich mehr Züge hinweg zumindest grob abzuschätzen. Mögliche Gründe dafür sind, dass sich der Zustand des Bretts mit jedem Zug nur selten grundlegend verändert. Es werden keine Steine verschoben und kein Stein wechselt seinen Zustand. Es kommen nur neue Steine hinzu und gelegentlich werden Steine geschlagen und entfernt. Mit jedem Zug[1] muss ein menschlicher Spieler

[1] egal ob real oder zur Stellungsanalyse virtuell im Kopf gespielt

somit nur ein lokal begrenztes Gebiet und ggf. die Implikationen auf umgebende Gebiete neu bewerten. Dem menschlichen Gehirn scheint dies recht leicht zu fallen, wohingegen dieser Ansatz für Computer nicht einfach nachmodellierbar ist.
Ein anderer denkbarer Grund für die Fähigkeit des menschlichen Gehirns das Spiel relativ weit abschätzen zu können, ist die stark visuelle Natur des Spiels. Es lassen sich Gebiete und wiederkehrende Formen, Muster und Zugabfolgen erkennen. Möglicherweise fällt es dem Gehirn leicht, solche visuellen Daten zu verarbeiten, zu abstrahieren, wiederzuerkennen und anzuwenden.
Ein sehr großes Problem ist die Schwierigkeit der Formalisierung von Wissen zur Programmierung einer Bewertungsfunktion. Wie schon erwähnt ist im Schach nur die Programmierung einer Funktion, die die Blätter des Alphabeta-Algorithmus bewertet, zwingend notwendig. Dies kann im einfachsten Fall durch eine mit den Figurenwerten gewichtete Abzählung des Materials[2] geschehen. Schrittweise lässt sich dies dann auch durch andere Heuristiken erweitern, aber zunächst reicht dieser Ansatz aus.
Im Go ist es weit schwieriger. Einen nennenswerten Materialunterschied gibt es meist nicht und andere von Computern einfach und schnell zu bewertende Kriterien wurden bisher noch nicht gefunden. Menschliche Spieler benutzen zur Bewertung einer Stellung viele abstrakte Konzepte, deren Beurteilung oft auf Erfahrungen basieren. Diese zu formalisieren und in einen Algorithmus zu „zwängen“ gestaltet sich zeitaufwändig und schwierig.

Bis heute wurden im Computer-Go immerhin Teilerfolge erzielt. Go auf sehr kleinen Brettern (5x5) wurde komplett gelöst ([24]) und Leben-und-Tod-Probleme sind oft lösbar ([27]). Die dafür verwendeten Methoden tragen aber nur wenig zur Lösung der obigen Probleme bei.
Die Spielstärke von Go-Programmen hat mittlerweile die Stärke durchschnittlicher Amateurspieler erreicht. Die Ansätze der unterschiedlichen Programme sind oft grundsätzlich verschieden, da es nicht wie im Schach einen Konsens über einen Grundalgorithmus gibt. Es gibt wissensbasierte Ansätze, Monte-Carlo-Algorithmen, Machine Learning-Ansätze und Programme, die den aus dem Schach bekannten Alpha-Beta-Algorithmus um geschicktes Pruning zu erweitern versuchen.
Das Niveau heutiger Programme wird sehr gut durch ein Spiel, das kürzlich zwischen dem koreanischen Profispieler Kim MyungWan und dem Programm MoGo stattfand ([16]), illustriert. MoGo lief in einer Mehrprozessor-Version auf dem Supercomputer „Hyugens“ mit 800 Kernen und bekam 7 Steine Handicap[3] und 90 Minuten Bedenkzeit. MyungWan brauchte nur ca. 20 Minuten seiner Bedenkzeit. MoGo gab schließlich nach 84 Minuten vernichtend geschlagen auf.
Was auf 19x19 Brettern nicht gelang, glückte zumindest auf 9x9. Im März 2008 schlug MoGo den Profi Catalin Taranu ohne Handicap ([1]). Doch auch dieser Sieg gelang nur auf einem Rechen-Cluster.

[2] die noch auf dem Brett befindlichen Figuren
[3] zu Handicap s. Abschnitt B.7

Go-KIs sind also auf alltäglicher Hardware und unter Turnierbedingungen (19x19-Bretter) noch weit entfernt von der Spielstärke starker Amateure und erst recht von Profispielern. Zumal sich auch, wie oben erwähnt wurde, noch kein aussichtsreicher Standardalgorithmus etabliert hat, ist Computer-Go definitiv ein interessantes Forschungsgebiet der Künstlichen Intelligenz.

1.2. Warum Maschinelles Lernen?

Es stellt sich nun die Frage, warum Maschinelles Lernen Programme dazu befähigen können sollte, auf einem höheren Niveau zu spielen.

Wie bereits erwähnt, sind reine A-Strategien aus Schach nicht einfach auf Go transferierbar. In Frage kommt also nur eine B-Strategie oder eine Mischform, also eine durch irgendeine Form von Wissen angereicherte A-Strategie. Wie schon am Beispiel der Blattevaluationsfunktionen erwähnt, ist die explizite Modellierung von Go-Wissen „per Hand“ schwierig, zeitintensiv und fehleranfällig. Bei der Integration eines neuen Konzepts läuft man stets Gefahr, dass Interaktionen mit anderen schon implementierten Konzepten auftreten und unbeabsichtigte Nebeneffekte bewirken ([19, Abschnitt 2.4.3. Quality and quantity of human knowledge]). Bspw. ist es möglich, dass eine neue Heuristik die Exploration einer bestimmten Art von Zügen ausschließt, obwohl man zuvor eine Heuristik geschrieben hatte, die genau diese Exploration sicherstellen sollte.
Ein menschlicher Spieler ist mit Erfahrung intuitiv fähig mehrere Konzepte in einer gegebenen Stellung gegeneinander abzuwägen. Einem Programm hingegen muss auch die Gewichtung beigebracht werden und diese ist nicht unbedingt (wie z.B. bei der Materialbewertung im Schach) eine Linearkombination verschiedener Faktoren. Die explizite Modellierung von Wissen steht also vor vielen Hindernissen und dennoch wurde dieser zeitaufwändige Weg bei einigen Programmen beschritten. Wissensbasierte Programme, z.B. The Many Faces of Go oder Go++, waren teilweise die stärksten ihrer Zeit ([26, Abschnitt Knowledge-based systems]).
Auch heutzutage gibt es wieder Arbeiten, in denen durch einen Experten modelliertes Wissen verwendet wird. Aktuell sind Programme, die den Monte-Carlo-Algorithmus UCT benutzen, Gegenstand der Forschung. Im Abschnitt 5.6.1 wird noch etwas genauer auf UCT eingegangen. In seiner grundlegenden Form versucht er durch Simulation zufälliger Fortsetzungen einer gegebenen Stellung ihren Wert abzuschätzen. Er stellt somit wieder einen Schritt in Richtung A-Strategie dar. Die Qualität seiner Abschätzung lässt sich verbessern[4], indem man in den Simulationen keine zufälligen Züge durchführt, sondern zur Zugauswahl Wissen benutzt und somit nur „sinnvolle“ Züge simuliert (vgl. [15]).
Sowohl das oben erwähnte MoGo, als auch Indigo verwenden UCT mit Expertenwissen ([2, 6]). In [6, Abschnitt Introduction] wird erwähnt, dass auf Expertenwissen basierende Zugpattern-Datenbanken anfällig für Fehler (Empfehlung schlechter Züge durch die Da-

[4] oder die zum Erreichen einer ausreichend genauen Abschätzung benötigte Zahl an Iterationen verkleinern

tenbank) und Löcher (es existieren Situationen, die von der Datenbank nicht abgedeckt werden) sind. Außerdem lässt sich aus den weiter oben schon erwähnten Gründen nur schwer neues Wissen formulieren bzw. neue geeignete Patterns finden und ohne Nebeneffekte in eine Patterndatenbank integrieren.
Daher erscheint ein Machine Learning Ansatz sinnvoll: Das Wissen soll automatisch aus einer Datenbank an Profispielerpartien gewonnen und ihr Wert statistisch verifiziert werden. Auf die Arbeit und Zeit eines Experten kann somit verzichtet werden. Die Korrektheit und Vollständigkeit der Datenbank soll (in einem gewissen Rahmen) automatisch erreicht werden.

1.3. Begriffsklärung

In diesem Abschnitt werden ein paar der in der Arbeit verwendeten Begriffe erklärt, damit dies nicht später in der Arbeit erfolgen muss.

(Trainings-)Beispiel

Der Algorithmus ist ein Supervised Learning Algorithmus, d.h. er bekommt eine Menge von Problemen und die jeweils zugehörigen Lösungen präsentiert und versucht, das hinter Problemen und Lösungen liegende Konzept zu lernen. Ein Paar aus Problem und Lösung heißt Trainingsbeispiel.
In diesem Fall ist das Problem eine realistische[5] Go-Stellung. Die Lösung ist der in dieser Stellung bestmögliche Zug. Da es bei einem derart komplexen Problem, wie dem Lösen einer Go-Stellung, schwer bis unmöglich ist, den besten Zug zu finden und ihn dem Algorithmus zu präsentieren, wird der beste Zug durch einen möglichst guten Zug angenähert.
Hierzu werden Spiele aus Go-Profispieler-Partien genommen. Als Lösung bekommt der Algorithmus den von den Profis gespielten Zug übergeben. Man kann annehmen, dass die Profi-Züge häufig mit den bestmöglichen Zügen übereinstimmen oder ihnen zumindest räumlich sehr nahe kommen.
Es ist aus zwei Gründen auch gar nicht nötig, den Algorithmus nur mit perfekten Zügen zu trainieren. Erstens ist das Ziel gar kein perfektes Spielen: Beim derzeitigen Stand von Computer-Go wäre es schon eine beachtliche Leistung eine KI zu programmieren, deren Stärke menschlichen Profis auch nur nahe kommt (vgl. Abschnitt 1.1). Zweitens muss dieser Algorithmus nicht auf sich allein gestellt arbeiten. Es reicht, wenn er nur zu einer gegebenen Stellung mehrere Zugvorschläge macht, die dann von einem nachfolgenden Algorithmus weiter verifiziert werden (s. Abschnitt 5.6.1).
Der Begriff des Beispiels ist eine Verallgemeinerung des Begriffs Trainingsbeispiel. Er bezeichnet ebenfalls ein Paar aus Go-Stellung und gespieltem Zug, bloß dass er im Gegensatz zum Trainingsbeispiel nicht im Training vorkommen muss. Z.B. kann in der Evaluation des Algorithmus von Testbeispielen gesprochen werden.

[5] d.h. in einem realen Spiel vorkommende

Training Set

Das Training Set ist die Gesamtheit aller Trainingsbeispiele, die dem Algorithmus im Lernvorgang übergeben werden, also die (aufbereitete) Datenbank der Profispielerpartien. Die Datenbank enthält mehrere hundert bis tausend Spiele und jedes Spiel enthält (auf 19x19 Brettern) im Schnitt über 200 Trainingsbeispiele.

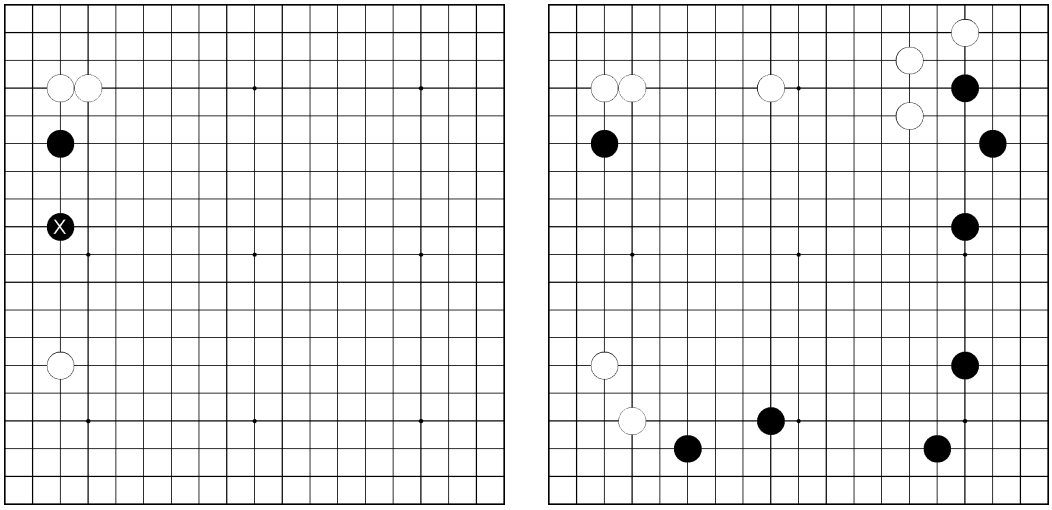
Move Guessing

Nachdem der Algorithmus mit Hilfe des Training Sets seine Patterndatenbank aufgebaut hat, ist er in der Lage Zugvorschläge für eine Stellung, die ihm übergeben wird, zu machen. Dieser Vorgang wird als Move Guessing bezeichnet, da der Algorithmus mit Hilfe seines Wissens versucht zu „erraten“, welchen Zug ein Profi in dieser Stellung spielen würde.
Der Programmteil, der das Move Guessing durchführt, wird als Move Guesser bezeichnet.

Patterns und Matching

Patterns lassen sich nach [19, Abschnitt 3.1.1. Definition of patterns] in drei Teile aufteilen: Die sog. Pattern Map, die ein paar der Felder eines Bretts enthält, der Pattern Context, der zusätzliche nichtlokale Informationen (z.B. ob ein Ko vorliegt) zur aktuellen Stellung enthält, sowie die Pattern Information, die bspw. den zu spielenden Zug vorgibt. Die Pattern Map sowie der Pattern Context werden in dieser Arbeit als Features bezeichnet. Was genau als Features gewählt wurde, wird in Abschnitt 3.2.1 erläutert.
Sowohl im Lernvorgang als auch später beim Move Guessing muss immer wieder geprüft werden, ob ein Pattern auf ein Brett passt. Dieser Vorgang wird als Matching bezeichnet. Es wird geprüft, ob das Brett alle vom Pattern vorgegebenen Features befriedigt.

In Abbildung 1.1 sieht man links ein Pattern. Zur Pattern Map gehören nur die Felder, die Steine enthalten, nicht die leeren Felder. Das mit X markierte Feld ist kein Stein, sondern der zu spielende Zug (die Pattern Information). Rechts sieht man ein Brett, auf das die Pattern Map offensichtlich zutrifft. Passen zusätzlich noch die anderen Features, der Pattern Context, so matcht das Pattern auf das Brett. Das Pattern gibt dann den nächsten Zug vor: das mit X markierte Feld.



(a) Pattern

(b) Stellung aus dem Finalspiel des 55. Honinbo-Turniers O Meien gegen Cho Sonjin

Abbildung 1.1.: ein Pattern und ein matchendes Brett

2. Zielsetzung und existierende Lösungsansätze

Das Ziel dieser Arbeit ist die Erstellung eines Algorithmus, der zu einer gegebenen Spielstellung mehrere Vorschläge, welche Züge er für vielversprechend hält, machen kann. Der Algorithmus muss also keineswegs nur einen einzigen Zug zurückliefern. Er stellt nur eine Vorstufe für weitere Algorithmen dar, die aus den erhaltenen Vorschlägen schließlich einen einzelnen Zug aussuchen.
Die Zugvorschläge soll der Algorithmus basierend auf Wissen, das er zuvor aus einer Spieldatenbank gewonnen hat, machen. Wichtig ist hierbei, dass das Wissen, wie in Abschnitt 1.2 gefordert, automatisch gewonnen wird und keinerlei Expertenwissen zur Generierung oder Evaluierung notwendig ist.
Ein Punkt, der auch von Interesse ist, ist, ob es gelingt, eine Art Eröffnungs- oder Joseki-Datenbank[1] aufzubauen. Mit Mittel- und Endspiel haben KIs im Go üblicherweise nicht so große Probleme, wie mit der Eröffnung. In der Eröffnung fällt es besonders schwer, die Auswirkungen von Zügen in die Zukunft abzuschätzen und daher wäre eine Patterndatenbank, die hier gute Ergebnisse liefert, von großem Wert.

Für Algorithmen, die zu einer gegebenen Stellung mehrere Zugvorschläge ausgeben, existieren bereits unterschiedlichste Ansätze, auf die im Folgenden kurz eingegangen wird.

2.1. Neuronale Netze

Eine Möglichkeit Wissen zu repräsentieren, ist die eher abstrakte Repräsentation in Form von Künstlichen Neuronalen Netzen. In Spielen können Neuronale Netze bspw. durch Spielen gegen sich selbst oder eine andere KI trainiert werden (Reinforcement Learning). In Backgammon hat sich dieser Ansatz als sehr erfolgreich herausgestellt ([23]). Go-Programme, die Neuronale Netze verwenden, sind z.B. Honte ([11]) und NeuroGo.
Das Training Neuronaler Netze ist zwar zeitaufwändig, aber sind sie einmal fertig trainiert, können sie schnell Ausgaben zu einem gestellten Problem liefern. Dies wäre z.B. bei einer Verwendung in UCT (s. Abschnitt 5.6.1) von Vorteil. Auch ihre kompakte Form ist hier von großem Vorteil. Ein Nachteil ist ihre schon erwähnte Abstraktheit und schlechte Interpretierbarkeit. Das ist in dieser Arbeit allerdings nicht relevant.
Neuronale Netze haben, wie auch in anderen Problemdomänen, damit zu kämpfen, dass

[1] Im Go gibt es nicht wie im Schach Bücher über Eröffnungen, da die Zahl der möglichen Variationen schon nach wenigen Zügen explodiert. Es gibt aber sehr wohl Bücher und Datenbanken über Zugabfolgen in der Ecke des Bretts, sog. Joseki, die sich im Laufe der Zeit etabliert haben.

die Eingabedaten geeignet aufbereitet werden müssen. Ein einfaches Weiterreichen der aktuellen Stellung an einen Input-Layer mit 19x19 Neuronen reicht bspw. nicht aus. Es ist also wieder nötig geeignetes Expertenwissen für die Aufbereitung der Rohdaten zu finden und es in algorithmischer Form zu modellieren.
Ein weiteres Problem ist folgendes: [5, Abschnitt 9.2. Temporal Differences(TD) Learning] zufolge eignen sich Neuronale Netze hauptsächlich dazu, positionelle Konzepte zu erfassen. Manchmal reicht es im Go aus nur positionell[2] zu beurteilen. In der Mehrzahl der Stellungen jedoch reicht positionelle Beurteilung nicht aus, da die zur Wahl stehenden Varianten expandiert werden müssen, um eine Wahl treffen zu können. In diesem Punkt unterscheiden sie sich allerdings kaum von auf Patterns basierenden Ansätzen, die ebenfalls nur auf Basis der aktuellen Situation bewerten.
Arbeiten, in denen Neuronale Netze in Go untersucht werden, sind [9, 12, 13, 18]. Ein Vergleich dieser Arbeiten ist nicht ohne Weiteres möglich, da sie zum Teil unterschiedliche Maße für die Beurteilung ihres Erfolgs benutzen.
[9] verwendet das gleiche Maß wie diese Arbeit (Definition s. Abschnitt 4.2). Die dort erzielten Genauigkeiten sind allerdings verglichen mit dieser Arbeit um ein Vielfaches kleiner. Grundsätzlich lässt sich auch erkennen, dass das Training Neuronaler Netze sehr zeitintensiv und daher vermutlich schlecht auf 19x19 Go skalierbar ist.

2.2. Explanation-Based Generalization / Learning

In [5, Abschnitt 9.5. Explanation-Based Generalization and Metaprogramming] wird Metaprogrammierung als möglicher Ansatz erwähnt. Das System analysiert taktische Zugabfolgen und generiert Regeln für diese Zugabfolgen in Prädikatenlogik. Die Ergebnisse sind zwar brauchbar, allerdings scheinen sie für den Zweck des Zugvorschlagens alleine nicht auszureichen. Sie eignen sich, um taktische Subziele im Spiel [3] zu lösen und den richtigen Zug in einer erzwungenen Zugabfolge zu finden. Für das Begreifen von abstrakten Konzepten zur Stellungsbeurteilung oder das Verfolgen übergeordneter Ziele scheinen sie allerdings nicht geeignet zu sein. Hier wäre eine Kombination mit einem anderen Ansatz nötig.
Problematisch für den Platzverbrauch und die Geschwindigkeit ist auch, dass sehr große Regelmengen erzeugt werden.

2.3. Patterns

Wie bereits in der Einleitung erwähnt, spielen menschliche Spieler häufig mit Hilfe von Mustererkennung. Menschen ziehen Muster zur positionellen Beurteilung von Teilen einer Stellung heran, Muster können direkt taktische Züge oder sogar ganze Zugabfolgen vorgeben und es gibt Literatur über Eröffnungszüge[4] und Eckzüge[5].

[2] z.B. nur gute oder schlechte Form
[3] z.B. Verbinden oder Trennen von Gruppen, Bilden oder Zerstören von Augen
[4] Fuseki
[5] Joseki

Es liegt also nahe die Verwendung von Mustern auch im Zusammenhang mit Go-KIs zu untersuchen.

Grundsätzlich bildet ein Muster einen Teil des Brettes ab und beinhaltet noch weitere Features, die direkt aus der Stellung abgelesen oder leicht ermittelt werden können. Es gibt zahlreiche Arbeiten, die das Aussehen, die Gewinnung oder die Verwendung von Patterns in anderen Algorithmen behandeln: [3, 4, 6, 10, 14, 15, 17, 22, 25] Diese Arbeit versucht viele der darin enthaltenen Ideen aufzugreifen.

In manchen Arbeiten (z.B. [14, 15]) werden Brettausschnitte fester Größe oder fester Form verwendet. Solche Repräsentationen haben den Vorteil, dass sie sich einfacher und effizienter kodieren lassen. Sie haben aber den Nachteil, dass sie den Anforderungen vieler Stellungen nicht gerecht werden. In einer Eröffnungssituation bspw. ist das Brett noch sehr leer. Nimmt man hier nur einen kleinen Ausschnitt des Bretts, enthält dieser möglicherweise keinen einzigen Stein und somit nahezu keine Information. In einer Endspielsituation hingegen ist es gut möglich, dass der feste Ausschnitt um den Zug herum sehr viele Steine enthält, sodass das später daraus erzeugte Pattern viel zu spezialisiert, schlecht abstrahiert und damit schlecht auf andere Stellungen anwendbar ist.
Daher orientiert sich diese Arbeit hauptsächlich an einer Arbeit, die eine flexiblere Darstellung verwendet: [6] benutzt die sog. K-Nearest-Neighbor Representation. Sie wählt aus den Steinen des Bretts k Steine aus, die dem Kreuzungspunkt, auf den der Zug gespielt wurde, am nächsten liegen. Ist die Stellung sehr voll, werden also nur Steine in unmittelbarer Nähe des Zugs gewählt. Ist sie eher leer, werden auch weiter entfernte Steine ausgewählt.
Die Hoffnung ist, dass diese Repräsentation flexibel genug ist, um nicht nur im Mittel- oder Endspiel Zugentscheidungen treffen zu können, sondern dass sie auch in der Eröffnung oder in Josekis gut spielen wird.
Die Bewertung der Güte der Patterns, sowie ein Teil der Patternfilterung sind ebenfalls aus [6] entnommen. Diese Arbeit weicht jedoch auch in vielen Punkten von [6] ab, z.B. indem mit variablem k gearbeitet und die Wahl des „richtigen“ k der Patternfilterung überlassen wird (näheres siehe Abschnitt 3.2.1 und 3.5). Außerdem sind zusätzliche Features, wie z.B. das Stage-Feature, in einem Pattern enthalten und die Güte des Move Guessers wird nicht integriert in einer KI sondern isoliert evaluiert. Insbesondere geht diese Arbeit auch über [6] hinaus, indem sie das Matching mit Hilfe von Bitboards (s. Abschnitt 3.3) einführt. Die Repräsentation über Bitboards ist sehr platzeffizient und der Matchingvorgang kann mit Hilfe nur einiger weniger Prozessoroperationen durchgeführt werden.

2.3.1. Evolutionäre Patternakquirierung

Ein Ansatz, der Patterns akquiriert und sich deutlich von anderen Patternakquirierungsalgorithmen unterscheidet, soll hier kurz erwähnt werden. Er wird in dieser Arbeit zwar nicht weiterverfolgt, ist aber dennoch zumindest interessant: In [17] werden Patterns mit Hilfe eines Evolutionären Algorithmus erzeugt.

Der Algorithmus fasst Patterns als Individuen einer Population auf. Er folgt zwar nicht streng dem normalen Ablauf eines Evolutionären Algorithmus (Selektion, Rekombination, Mutation, Bewertung). Sein Ziel ist auch nicht das Finden eines an das Problem am besten angepassten Individuums, sondern das Finden einer bestangepassten Population. Der Algorithmus ist allerdings eine Analogie zu einem sich anpassenden ökologischen System und kann somit als Evolutionärer Algorithmus gelten.

```
step <- 1
while step < laststep
  choose a datum from a training set
  if no rule matches the datum
    then make a new rule
    else feed matched rules
  for all rules
    if activation of a rule > threshold
      then split the rule
    activation of a rule <- activation of a rule - 1
    if activation of a rule = 0 then the rule dies
  end
  step <- step + 1
end
```

Ein Pattern (bzw. oben eine Regel) stellt gewisse Bedingungen an die Felder eines Bretts. Das Brett matcht, wenn all diese Bedingungen erfüllt sind. Der Algorithmus präsentiert einer Pattern-Population Nahrung in Form von Trainingsbeispielen. Jedes Mal, wenn ein Pattern auf ein Stück Nahrung matcht, konsumiert das Pattern diese Nahrung. Hat ein Pattern öfter als ein gewisser Schwellwert Nahrung konsumiert, teilt es sich in mehrere Patterns, die Verfeinerungen des ursprünglichen Patterns darstellen. Die Verfeinerung geschieht, indem dem Pattern eine neue Bedingung hinzugefügt wird. Durch die Teilungen bildet der Algorithmus einen Baum aus sich immer weiter spezialisierenden Patterns.
Patterns spalten sich nicht nur, sondern sterben auch, wenn sie zu lange keine Nahrung konsumieren. Dadurch werden manche Äste des wachsenden Baums wieder beschnitten. Zum Schluss ist er in Teilbäumen, die für viele Stellungen relevant sind, groß und weit verzweigt und in weniger relevanten Bereichen weniger verzweigt.

Mit diesem Ansatz ist es gelungen eine Vielzahl sehr unterschiedlich gearteter Patterns, z.B. Patterns für lokal begrenzte Konstellationen, aber auch Ganzbrettmuster, zu gewinnen.
Zur Zeit der Veröffentlichung dieses Algorithmus gab es noch keine einfache Möglichkeit der Integration dieses Algorithmus in einen Spielalgorithmus. Außerdem wurde die Güte der vom Algorithmus gewonnenen Patterns nicht objektiv beurteilt. Es wurde lediglich ein Teil der Patterns einem guten Go-Spieler vorgelegt, damit dieser ihre Qualität beurteilt. Für eine aussagekräftige Evaluation hätten die Patterns in eine Art Move Guesser

integriert und seine Vorhersagegenauigkeit o.ä. evaluiert werden müssen.

3. Funktionsweise der Patternakquirierung und des Move Guessers

Das Programm besteht aus zwei Teilen, dem Lernalgorithmus und dem Move Guesser. Der Lernalgorithmus versucht aus einer Spieldatenbank, dem Training Set, Patterns zu lernen und eine Patterndatenbank aufzubauen, die später den Move Guesser befähigen soll, in einer gegebenen Stellung einen Zugvorschlag zu machen.
Der Lernalgorithmus gliedert sich in drei Schritte. Im ersten werden die Spiele aus der Spieldatenbank ausgelesen, in ein Format geparst, das das Programm versteht und ungeeignete Spiele ausgefiltert. Aus den verbleibenden Spielen werden die Trainingsbeispiele extrahiert und normalisiert. Im zweiten Schritt werden aus den Trainingsbeispielen Features extrahiert und daraus Zugpattterns generiert. Im dritten Schritt werden die Patterns bewertet und bekommen diesen Wert, der ihre Anwendbarkeit bzw. ihren Erfolg in Spielen repräsentiert, zugewiesen.

Nachdem die Patterndatenbank aufgebaut ist, kann sie für ihren vorgesehenen Zweck, das Move Guessing, verwendet werden. Hierbei bekommt das Programm eine Stellung gegeben, durchsucht die Patterndatenbank nach passenden Patterns und wählt diejenigen, die die höchsten Werte besitzen, aus.

3.1. Trainingsbeispiel-Gewinnung

3.1.1. Parsen und Filtern der Datenbank

Zunächst müssen die Spiele aus der Spieldatenbank extrahiert werden. Ein für Go-Datenbanken weit verbreitetes Format ist das Smart Game Format[1]. SGF-Dateien enthalten u.a. die gespielten Züge, Informationen über die Stärke beider Spieler (s. Abschnitt B.7) sowie den Spielausgang (gewonnen durch Zeit/durch Aufgabe/nach Punkten).

Die SGF-Dateien der Datenbank werden eingelesen und geparst. Anschließend werden unpassende Spiele ausgefiltert. In 9x9-Go gab es z.B. nicht genügend Profi-Partien und daher musste auch auf Amateurpartien zurückgegriffen werden. Daher wurden Spiele ausgefiltert, in denen das Niveau mind. eines der beiden Spieler unterhalb einer gewissen Grenze (z.B. 5k) lag. Außerdem wurden generell Spiele ausgefiltert, die mit einem zu schlechten Ergebnis für einen Spieler ausgegangen sind (bspw. eine Niederlage mit mehr als 6.5 Punkten Vorsprung für den Sieger). Ein zu schlechtes Ergebnis kann ein Indiz für

[1] Die vollständige Spezifikation findet man unter `http://www.red-bean.com/sgf/`

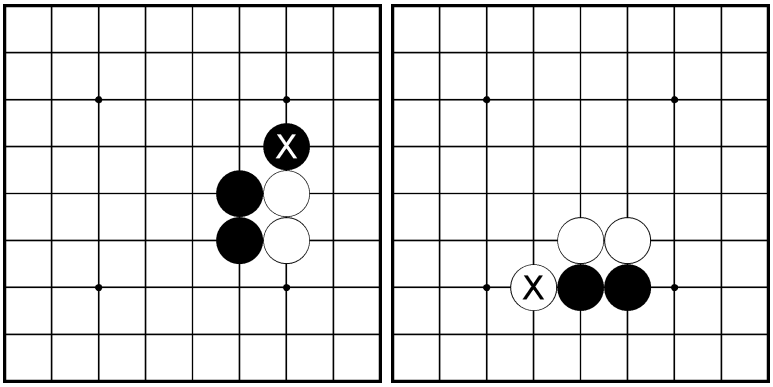


Abbildung 3.1.: zwei äquivalente Stellungen

einen dramatischen Fehler des Verlierers sein. Einen solchen Fehler soll das Programm nicht lernen.

3.1.2. Extraktion der Trainingsbeispiele und Preprocessing

Nach dem Parsen und Filtern der Spiele wird jedes Spiel Zug für Zug nachgespielt und nach jedem Zug die aktuelle Stellung mit dem nächsten Zug zu einem Trainingsbeispiel zusammengruppiert.
Bevor die Trainingsbeispiele dem eigentlichen Lernalgorithmus übergeben werden, werden sie noch bezüglich Brettrotationen und Inversion der schwarzen und weißen Steine normiert. Das Beispiel in Abbildung 3.1 soll dies verdeutlichen. Der mit X markierte Stein ist der Zug, der in dieser Situation von den Profis gespielt wurde. Die zweite Stellung ist, wenn man die Farben invertiert und das Brett an seiner Hauptdiagonalen spiegelt, identisch zur ersten Stellung. Die beiden Stellungen sind also von ihrer Aussage oder ihrer Bedeutung her äquivalent zueinander.
Das Brett lässt sich, wie in Abbildung 3.2 zu sehen, in acht Teile unterteilen. Liegt der Zug eines Trainingsbeispiels nicht in Bereich 2, wird das Trainingsbeispiel durch elementare Brettspiegelungen (Spiegelung an der Waagerechten, Senkrechten oder Nebendiagonalen) so gespiegelt, dass der Zug in diesem Bereich zum Liegen kommt[2]. Außerdem wird, im Falle dass der Zug des Trainingsbeispiels von Weiß gespielt wurde, die Farbe invertiert, damit alle Trainingsbeispiele aus der Sicht von Schwarz gespielt sind.
Das zweite Brett in Abbildung 3.1 würde bei der Normierung erst farbinvertiert und dann an der Nebendiagonalen, der Waagerechten und der Senkrechten gespiegelt. Daraus resultiert die Stellung des ersten Bretts. Man könnte die Stellung zwar schneller normieren, wenn man sie an der Hauptdiagonalen spiegeln würde, allerdings reichen die anderen drei Elementarspiegelungen aus, um von jedem beliebigen in jedes andere belie-

[2]Die Festlegung auf Bereich 2 ist willkürlich. Es hätte auch jeder der anderen sieben gewählt werden können. Der 2. wurde gewählt, weil traditionell der erste Eröffnungszug eines Spiels in diesen Bereich gespielt wird.

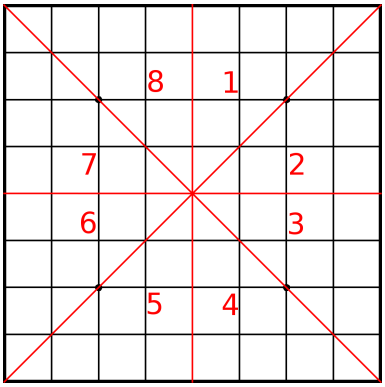


Abbildung 3.2.: die Teile des Bretts

bige Achtel zu rotieren. Daher beschränkt sich das Programm auf diese drei Spiegelungen und nimmt eventuellen Zeitverlust in Kauf. Bei der Rotation von einem Achtel in ein anderes muss beachtet werden, dass die Reihenfolge der Elementarspiegelungen nicht beliebig vertauschbar ist. Die Reihenfolge Waagerechte, Nebendiagonale, Senkrechte bspw. würde zu einem falschen Ergebnis führen.

3.2. Featureextraktion & Patterngenerierung

3.2.1. Featureextraktion

In der Featureextraktion muss der Algorithmus aus einem übergebenen Trainingsbeispiel diejenigen Features ermitteln, die er für interessant bzw. wichtig hält.

Die verwendeten Features sind:

- ein Ausschnitt des Bretts
- befindet sich der gespielte Zug in der Nähe eines Brettrandes oder einer Ecke?
- liegt in der Stellung eine Ko-Stellung vor?
- die Stage (z.B. Eröffnung, Mittelspiel, Endspiel), in der sich das Spiel gerade befindet

Wie in Abschnitt 1.3 beschrieben, bezeichnet man den Brettausschnitt auch als Pattern Map und die restlichen Features als Pattern Context.

Die Anwesenheit von Bretträndern oder Ecken habe ich in den Pattern Context aufgenommen, da es im Go von großer Bedeutung ist, ob sich ein Zug in der Nähe eines Randes oder einer Ecke befindet. Züge in der Mitte des Bretts sind in der Regel auch an

vielen anderen Stellen in der Brettmitte anwendbar, aber Züge an den Rändern oder in den Ecken ergeben in den meisten Fällen nur genau dort, wo sie auch aufgetreten sind, Sinn. Das geht so weit, dass Züge auf der dritten Linie[3] oft nicht einmal auf der vierten Linie anwendbar sind.

Die Anwesenheit einer Ko-Stellung wurde in die Features aufgenommen, da menschliche Spieler in Ko-Stellungen Züge spielen, die anderenfalls nicht gespielt worden wären. Dies liegt daran, dass man manche Drohungen im Laufe des Spiels nicht spielt und sie sich für einen sog. Ko-Kampf aufhebt. Im Beispiel-Spiel in Abschnitt B sieht man ab Zug 66 einen solchen Kampf.

Das Stage-Feature wurde ursprünglich hinzugefügt, um die Vorhersagegenauigkeit zu erhöhen. In einem realen Spiel gibt es viele Züge, die hauptsächlich in einer Stage auftauchen und kaum auf andere Stages übertragbar sind. Daher sollte der Move Guesser z.B. keine Endspielzüge bereits im Mittelspiel vorschlagen.
Durch das Stage-Feature wird das Training Set quasi in mehrere Teile unterteilt. Jeder Teil enthält nur die Trainingsbeispiele einer bestimmten Phase des Spiels. Die aus den Beispielen generierten Patterns tragen dann die Information, aus welcher Stage sie stammen. Im Matchingvorgang können auf eine gegebene Stellung lediglich Patterns, die zur selben Stage wie die gegebene Stellung gehören, matchen. In meinem Programm habe ich mit fünf Stages gearbeitet. Jedes Spiel wird also in fünf Abschnitte unterteilt (bspw. auf 9x9: Zug 1 bis 15, 16 bis 30, 31 bis 45, 46 bis 60, 61 und größer). Insbesondere auf 9x9 war die Menge der zur Verfügung stehenden Trainingsbeispiele relativ gering, sodass eine feinere Unterteilung als in fünf Stages dazu geführt hätte, dass die für jede Stage zur Verfügung stehende Zahl an Trainingsbeispielen zu gering geworden wäre, um einen guten Move Guesser zu trainieren.
In der Auswertung stellte sich heraus, dass das Stage-Feature keine Verbesserung, sondern eine leichte Verschlechterung mit sich brachte (s. Abschnitt 4.7). Dies resultiert wahrscheinlich daher, dass das für jede Stage zur Verfügung stehende Training Set relativ klein war. Das Feature wurde dennoch im Programm belassen, da der Move Guesser mit diesem Feature zu einer gegebenen Stellung nur noch die Patterns der zugehörigen Stage untersuchen muss, was einen sehr großen Geschwindigkeitsvorteil mit sich bringt.

Die Pattern Map (der zu betrachtende Brettausschnitt) wird folgendermaßen ermittelt: Interessant ist der Bereich des Bretts, der sich unmittelbar um den gespielten Zug herum befindet, da es wichtig zu wissen ist, in welchem lokalen Kontext ein Zug gespielt wurde, um beurteilen zu können, wie sinnvoll er war. Die K-Nearest-Neighbor Representation wählt aus den Steinen des Bretts k Steine, die dem gespielten Zug am nächsten liegen, aus. Außerdem können nicht nur k Steine sondern z.B. auch $k-1$ Steine und ein

[3]Im Go gibt es oft keine Unterscheidung zw. Zeilen und Spalten. In Go-Literatur wird deshalb meist neutral z.B. von „3rd line“ gesprochen, um einen Zug, der zwei Kreuzungen vom Rand entfernt ist, zu bezeichnen.

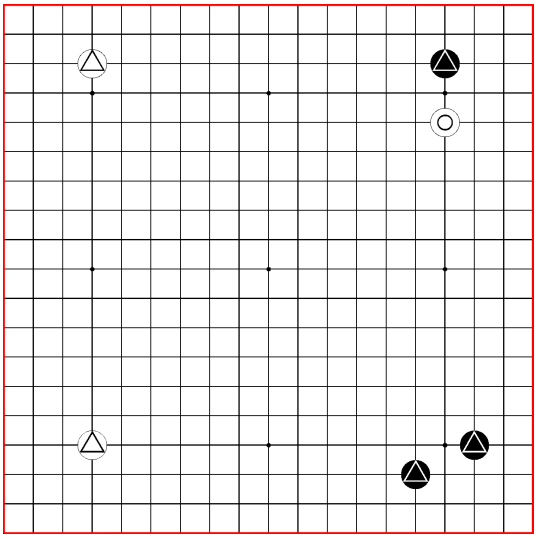


Abbildung 3.3.: Ein Eröffnungszug aus einem Spiel zwischen Masayuki und Chikun; Das Beispiel wurde hier nicht farbnormiert (also alle Steine farbinvertiert, damit der Zug von Schwarz gespielt wird), im Programm würde dies aber geschehen.

Brettrand oder $k-2$ Steine und zwei Bretträndern (also eine Ecke) ausgewählt werden. Die Ränder haben später im Zusammenhang mit Translationen (s. Abschnitt 3.2.2) eine Bedeutung: Wenn eine Ecke selektiert wird, kann das resultierende Pattern nicht mehr frei auf dem Brett verschoben werden. Es muss zwingend in einer Ecke liegen.

In Abbildung 3.3 sieht man eine Eröffnungssituation. Der gespielte Zug ist mit einem Kreis markiert. Mit k=9 werden die fünf mit Dreiecken markierten Steine und die vier rot markierten Ränder selektiert und für die K-Nearest-Neighbor Representation verwendet.

In Abbildung 3.4 links wählt der Algorithmus für k=7 die fünf markierten Steine und die zwei rot markierten Bretträndern.

In Abbildung 3.4 ist rechts eine Situation zu sehen, in der der Algorithmus nur noch sehr lokal begrenzte Steine auswählt.

In diesen drei Beispielen wurden unterschiedliche Werte für k verwendet. Auch im Algorithmus wird kein festes k gewählt. Aus jedem Trainingsbeispiel extrahiert der Algorithmus je ein Pattern für k von 4 bis k_{max} (in den Tests ist k_{max} meist 9). Im Pattern Filtering (s. Abschnitt 3.5) sollen u.a. diejenigen Patterns mit zu hohem k (zu spezialisierte Patterns) durch Support Filtering eliminiert werden und in der Patternbewertung (s. Abschnitt 3.4) die Patterns mit zu kleinem k (zu allgemeine Patterns) eine schlechte Bewertung bekommen und durch Confidence Filtering ausgefiltert werden.

In den gewählten Brettausschnitten (und damit später auch in den Patterns) sind grundsätzlich nur Steine enthalten. Es werden keine leeren Felder in das Pattern aufgenommen,

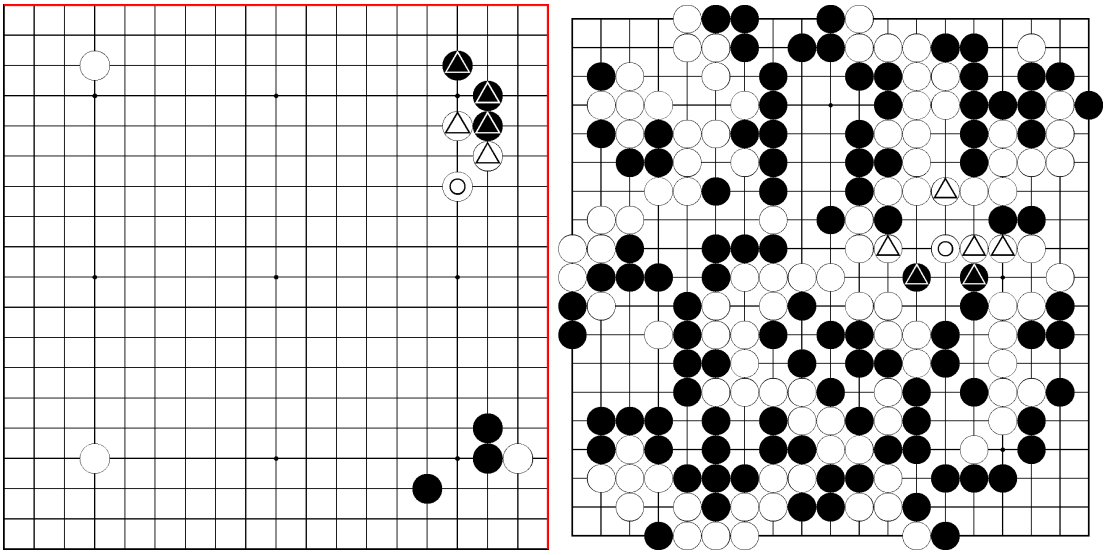


Abbildung 3.4.: Ein Joseki des selben Spiels, sowie ein Zug aus dem frühen Endspiel

also vorgeschrieben, dass ein Brett, auf das das Pattern matchen soll, an dieser Stelle ein freies Feld haben muss.
Es gibt zwei Gründe dafür: Erstens ist es bei freien Feldern (im Gegensatz zu besetzten Feldern) nicht so leicht zu entscheiden, ob sie von Bedeutung sind oder nicht. Für manche Patterns ist es irrelevant, ob ein Feld in der Nähe frei oder belegt ist, für andere Patterns hingegen sind freie Felder von großer Bedeutung. Zweitens sind freie Felder in Patterns implizit enthalten. Ein Pattern enthält implizit an einer Stelle ein freies Feld, wenn es dort keinen Stein besitzt. Wäre im Trainingsbeispiel, aus dem dieses Pattern generiert wurde, an der betreffenden Stelle ein Stein gewesen, so wäre er in das Pattern aufgenommen worden. Das Pattern wäre somit spezieller gewesen und hätte vermutlich in der Evaluation (Abschnitt 3.4) eine höhere Genauigkeit erhalten und bekäme bei der Zugvorhersage den Vorzug gegenüber dem Pattern, das den Stein nicht enthält. Der Umstand, dass das Pattern, das an dieser Stelle nichts enthält, in der Vorhersage zum Zug kommt, deutet also darauf hin, dass es vorher kein besseres Pattern gab, also insbesondere auch keins, das den zusätzlichen Stein enthält.

Es ist gut möglich, dass das Aufnehmen weiterer Features sinnvoll gewesen wäre. In [25] wird bspw. noch die Zahl der Freiheiten, die ein Stein, der so gespielt würde, wie es das Pattern vorschlägt, hätte, erwähnt. Auch wird die Entfernung des vom Pattern vorgeschlagenen Zugs zum vorangegangenen Zug erwähnt. Dieses Feature würde häufig dazu führen, dass der Move Guesser Züge vorschlägt, die in der Nähe des vorherigen gegnerischen Zugs liegen. Dies entspricht oft auch menschlicher Spielweise.
In [10] taucht noch eine Vielzahl anderer Features auf. Eine größere Zahl an Features würde jedoch dazu führen, dass die Patterns spezieller würden und eine größere Anzahl

an Patterns generiert würde. Die Patterndatenbank sprengt jedoch ohnehin schon den durch den verfügbaren Arbeitsspeicher vorgegebenen Rahmen, weshalb auf die Integration weiterer Features verzichtet wurde (vgl. hierzu auch 4.10).

3.2.2. Rotationen & Translationen

Die Trainingsbeispiele wurden, wie oben erwähnt, vor der Featureextraktion bzgl. Rotation und Farbe normiert. Später bei der Zugvorhersage müssen die Patterns auch auf nicht normierte Bretter gematcht werden können.
Die Farbinvertierung der Patterns stellt hierbei kein großes Problem dar. Sie kann ggf. live beim Matching geschehen. Wie in Abschnitt 3.3 erklärt wird, bestehen die fertigen Patterns aus je einem Bitboard für den schwarzen und einem für den weißen Spieler. Muss beim Matching farbinvertiert werden, werden nur die beiden Bitboards vertauscht.
Wollte man die Rotation der Patterns auch live beim Matching durchführen, so müsste das Pattern in alle acht möglichen Rotationen rotiert werden und geprüft werden, ob eine davon auf das Brett matcht. Rotieren ist vergleichsweise zeitaufwändig. Daher wurde die Möglichkeit gewählt, zusätzlich zur normierten Rotation auch die sieben weiteren Rotationen im Lernvorgang zu berechnen und alle acht Rotationen gemeinsam abzuspeichern. Die acht Rotationen heißen dann Elementarpattern und zusammengruppiert heißen sie Pattern.
Translationen müssen ebenfalls beachtet werden. Ein Pattern ist oft nicht nur an einer festen Position korrekt[4], sondern auch in diversen Translationen. Oft passiert es jedoch, dass viele Translationen eines Patterns sinnvoll wären, das Pattern aber nicht in allen diesen Translationen im Training Set auftaucht. Es erscheint daher sinnvoll, sich nicht darauf zu verlassen, dass die Translationen ebenfalls im Training Set auftauchen und somit automatisch gelernt werden, sondern sie, wie die Rotationen, vorher zu erzeugen, ebenfalls als Elementarpattern zu behandeln und sie gemeinsam mit der Standardkonfiguration abzuspeichern.
Aus Platzgründen werden die Translationen jedoch nicht wie die Rotationen vorab berechnet. Stattdessen wird in einem Pattern nur abgespeichert, welche Translationen möglich sind, d.h. welche minimalen und maximalen Translationen erlaubt sind. Die Translationen werden dann live beim Matching berechnet. Die Berechnung der Translationen geschieht durch einfache Bitshifts der Bitboards (s. Abschnitt 3.3).
Die Bestimmung der minimal und maximal erlaubten Translationen geschieht wie folgt: Wurden bei der Featureextraktion zwei Brettkanten gewählt, darf das Pattern nur in der Ecke auftreten. Translationen sind in diesem Fall also gar nicht möglich. Wurde genau eine Brettkante ausgewählt, ist das Pattern nur entlang dieses Randes verschiebbar. Wurde keine Brettkante gewählt, so ist das Pattern aber dennoch nicht komplett frei auf dem Brett verschiebbar. Das Pattern wird probeweise auf alle Beispiele des Training Sets gematcht und es wird geschaut, in welchen minimalen und maximalen Translationen das Pattern auftritt.
In Abbildung 3.5 sieht man ein Pattern, das im ersten Trainingsbeispiel unten (der

[4]Korrekt bedeutet hier, dass der vom Pattern vorgeschlagene Zug Sinn ergibt und im Training Set auftauchen könnte.

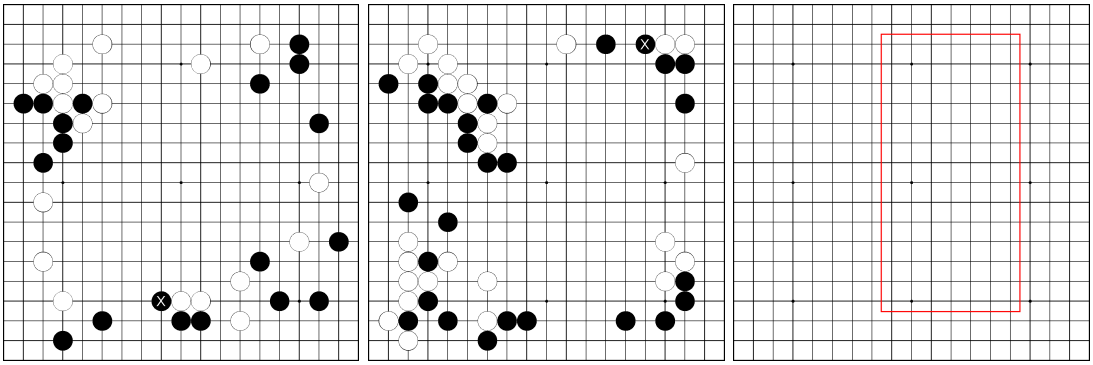


Abbildung 3.5.: minimales und maximales Auftreten eines Patterns im Training Set

zu spielende Zug ist mit X markiert) und im zweiten rechts oben auftritt. Die beiden Auftrittspunkte der Beispiele definieren ein Rechteck, das im dritten Bild zu sehen ist. Die innerhalb dieses Rechtecks möglichen Translationen werden in Form von Werten für die minimale und maximale Translation im Pattern abgespeichert.

3.3. Bitboards

Im Laufe der Entwicklung sind massive Probleme mit dem Speicherverbrauch und der Geschwindigkeit des Algorithmus aufgetreten. Die Darstellung der Patterns wurde daher von zweidimensionalen Stein-Arrays auf sog. Bitboards umgestellt. Analog zu den Patterns wurde auch die Repräsentation der Bretter auf Bitboards umgestellt, damit das Matching effizienter durchgeführt werden kann.
Bitboards sind ursprünglich aus anderen Brettspielen, wie Dame oder Schach bekannt (s. [20]). Für jede Klasse an Figuren (z.B. schwarze Läufer) benötigt man hier eine 64-Bit-Zahl. Jedes der 64 Bit korrespondiert mit einem der 64 Felder des Bretts. Steht auf einem Feld eine Figur, ist das zugehörige Bit der zur Figurenklasse gehörenden Zahl gesetzt, andernfalls bleibt es 0. Im Schach braucht man also 6 (für die 6 unterschiedlichen Figuren) $\cdot$ 2 (für Schwarz und Weiß) 64 Bit-Zahlen, um das komplette Brett zu repräsentieren. Operationen auf dem Brett werden nur noch durch Bitoperationen realisiert. Z.B. kann ein Vorziehen der Bauern durch einen Bitshift um 8 Bits oder das Prüfen, ob eine Figur geschlagen wurde, durch eine and-Operation realisiert werden. Problematisch an Bitboards ist eine erschwerte Programmier- und Debugbarkeit.
Im Go sind die Bitboards einfacher, weil nur zwei Bitboards (für die schwarzen und die weißen Steine) gepflegt werden müssen. Allerdings sind sie größer, weil das gesamte Brett nicht in eine sondern für 9x9-Bretter nur in zwei und für 19x19-Bretter in sechs 64-Bit-Zahlen passt.
Mit Bitboards sind auch die Farbinversionen, die live beim Matching stattfinden, schnell durchführbar, indem man einfach das weiße mit dem schwarzen Bitboard vertauscht.
Da die Bitboards nicht nur für Bretter, sondern auch für Patterns verwendet werden, be-

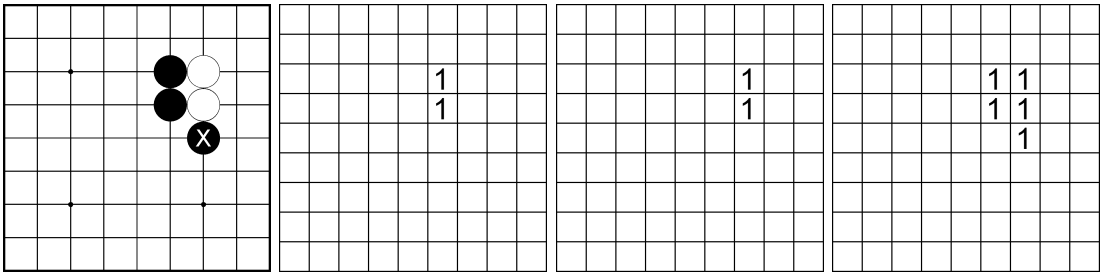


Abbildung 3.6.: Das ursprüngliche Pattern, das resultierende schwarze Bitboard, das weiße Bitboard und die Maske

steht an die Bitboards noch die Anforderung, dass sie Informationen enthalten müssen, welche Felder des Bretts für das Pattern relevant sind. Ein Pattern interessiert beim Matching nur, dass das Brett, auf das es gematcht werden soll, die gleichen Steine an den gleichen Positionen wie es selbst enthält. Welche Steine auf den sonstigen Feldern liegen, ist irrelevant. Es muss also eine Art Wegmaskieren von irrelevanten Feldern möglich sein.
Da die Darstellung von Bitboards in mehreren 64-Bit-Zahlen schwer nachvollziehbar ist, stelle ich sie im Folgenden in der äquivalenten Darstellung als zweidimensionales Bitarray dar. Die lineare Form als Zahl erhält man daraus durch Aneinanderreihen der Zeilen des zweidimensionalen Arrays. Außerdem lasse ich der Übersichtlichkeit halber alle 0-Bits weg.
In Abbildung 3.6 sieht man zunächst ein Pattern. Der mit X markierte Punkt ist der, auf den Schwarz diesem Pattern zufolge spielen soll. Daneben sieht man die Bitboards für die schwarzen und weißen Steine, deren Aussehen selbsterklärend sein sollte. Zu beachten ist nur, dass im schwarzen Bitboard das Feld unterhalb der weißen Steine (das X-Feld) nicht auf 1, sondern auf 0 gesetzt ist. Hier darf noch kein Stein liegen, da Schwarz erst noch auf diesen Punkt spielen soll. Zuletzt sieht man die Maske. Sie ist auf allen für das Pattern relevanten Feldern auf 1 gesetzt. Das X-Feld ist ebenfalls auf 1 gesetzt, da es für das Pattern relevant ist, welchen Inhalt, nämlich „leer“ (bzw. 0 im schwarzen und im weißen Bitboard), dieses Feld hat.
Bretter enthalten, wie bereits erwähnt, ebenfalls ein schwarzes und ein weißes Bitboard, allerdings keine Maske. Der Matchingvorgang geht nun folgendermaßen vonstatten: Beide Bitboards des Bretts werden jeweils mit der Maske per binärer and-Operation verknüpft. Dadurch fallen die Bits weg, die in der Maske nicht auf 1 gesetzt (also irrelevant) sind. Wenn (und nur wenn) die and-Verknüpfung des schwarzen Brett-Bitboards mit der Maske identisch mit dem schwarzen Pattern-Bitboard und die and-Verknüpfung des weißen Brett-Bitboards mit der Maske identisch mit dem weißen Pattern-Bitboard ist, matcht das Pattern das gegebene Brett.
In Abbildung 3.7 sieht man ein Brett und die zugehörigen Bitboards. Es soll geprüft werden, ob das Pattern aus Abbildung 3.6 matcht. Man kann sich leicht davon überzeugen, dass schwarzes Brett-Bitboard & Maske = schwarzes Pattern-Bitboard gilt. Selbiges gilt

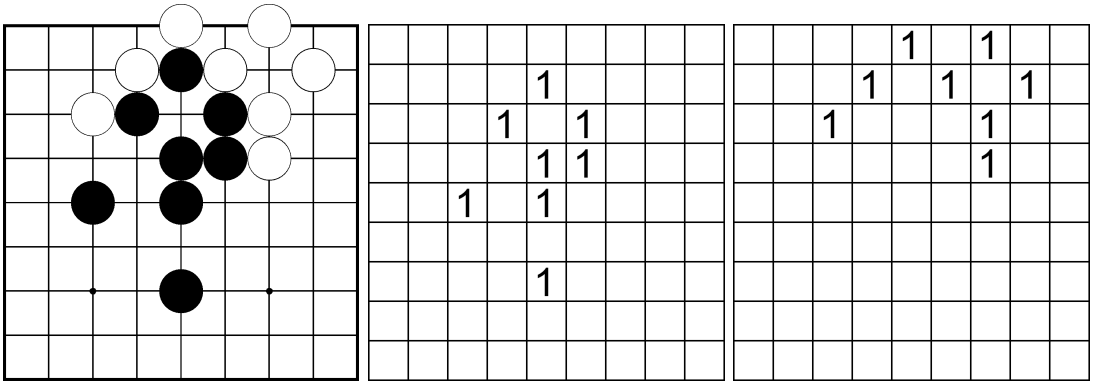


Abbildung 3.7.: Ein Brett, das schwarze und das weiße Bitboard

für Weiß. Das Pattern matcht also.
Wie bereits in Abschnitt 3.2.2 erwähnt, eignen sich die Bitboards auch sehr gut, um die Patterntranslationen durchzuführen. Soll ein Pattern um Δx in x- und Δy in y-Richtung verschoben werden, so braucht hierzu auf den Bitboards und der Maske des Patterns nur ein Bitshift um $19\Delta y + \Delta x$ Bits (bzw. $9\Delta y + \Delta x$ Bits für 9x9-Bretter) durchgeführt werden.
Die Darstellung der Bitboards ist sehr kompakt. Ein Pattern braucht für 19x19 Go lediglich 3 (schwarz, weiß, Maske) $\cdot$ 6 $\cdot$ 64 Bit. Der Matching-Prozess muss keine aufwändige Datenstruktur mehr durchforsten und mit dem Brett vergleichen. Er braucht nun 2 $\cdot$ 6 and-Operationen und ebenso viele Vergleichsoperationen. Die erstmalige Erstellung der Bitboards ist möglicherweise etwas aufwändiger, allerdings geschieht sie nur einmal für jedes Pattern. Der Matching-Prozess hingegen wird im Lernprozess (#Patterns $\cdot$ #Trainingsbeispiele im Training Set) mal durchgeführt. Auch später beim Move Guessing muss das Matching sehr schnell sein, damit das Vorschlagen von Zügen nicht viel Zeit in Anspruch nimmt.
Die Bretter konnten leider nicht absolut konsequent auf Bitboards umgestellt werden. Das Problem ist, dass das UCT-Modul (s. Abschnitt 5.6.1) sehr schnell Züge auf dem Brett ausführen können muss und hierfür stellten sich Bitboards als weniger geeignet heraus. Stattdessen gibt es dafür geeignetere, auf Go spezialisierte Darstellungen[5]. Daher enthalten die Bretter intern beide Repräsentationen. Wird ein Zug ausgeführt, so geschieht dies zunächst in der dafür geeigneten Repräsentation und die veränderten Felder werden anschließend in den Bitboards nachgepflegt. Aus Effizienzgründen lassen sich die Bitboards in UCT auch abschalten. Dann kann allerdings der Move Guesser nicht mehr befragt werden.

[5] In solchen Darstellungen muss schnell ermittelbar sein, welche Steine zusammen zu einer Kette gehören, wie viele Freiheiten eine Kette hat und ob sie evtl. durch einen Zug geschlagen worden ist.

3.4. Patternevaluation

Nachdem die Patterns aus dem Training Set erlernt wurden, müssen sie noch bewertet werden.
Ein zu bewertendes Pattern wird mit jedem Trainingsbeispiel des Training Sets verglichen. Dabei wird mitgezählt, wie oft das Pattern matcht und wie oft der Zug, den es vorschlägt, im Falle dass es matcht, auch tatsächlich gespielt wurde. Der Wert des Patterns ergibt sich dann zu $V = P(\text{gespielt}|\text{match}) = \frac{N(\text{gespielt} \cap \text{match})}{N(\text{match})}$ Der Wert spiegelt damit die Genauigkeit des Patterns, wie verlässlich der von ihm vorgeschlagene Zug ist, wieder.
Dieser Wert ist später im Move Guessing gut brauchbar: Bekommt der Move Guesser eine Stellung übergeben, durchsucht er die Patterndatenbank nach Patterns, die matchen. Von den matchenden Patterns liefert er dann den Zugvorschlag des Patterns mit der höchsten Genauigkeit zurück. Ist das Vorschlagen mehrerer Züge gewünscht, kann er auch die Vorschläge der m genauesten Patterns zurück liefern.
Diese Vorgehen ist zwar sehr intuitiv, berücksichtigt aber ein paar Effekte nicht. Der Algorithmus nimmt an, dass es nur einen Grund dafür geben kann, wenn ein Pattern matcht und dennoch nicht gespielt wird: Das Pattern ist unakkurat. Es gibt jedoch auch die Möglichkeit, dass das Pattern zwar gespielt werden könnte, also sehr wohl akkurat ist, nur dass im Trainingsbeispiel in dem Moment andere Dinge Vorrang hatten. Ein Zug wird immer aus einer Vielzahl an Überlegungen heraus gespielt. Mehrere Muster können auch miteinander interagieren (manchmal verstärkend, manchmal konkurrierend). All dies wird von diesem Vorgehen nicht berücksichtigt. Man müsste also in der Berechnung der Patternwerte mindestens berücksichtigen, dass Patterns auch positiv bewertet werden sollten, wenn sie in einem Trainingsbeispiel matchen und erst in der Zukunft gespielt werden.

3.5. Pattern Filtering

Da in der Patterngenerierung sehr viele Patterns erzeugt werden, wird nun versucht, aus diesen die nützlichsten herauszufiltern. Dies geschieht analog zum Support/Confidence Filtering beim Lernen von Assoziationsregeln.
Eine Assoziationsregel beschreibt einen Zusammenhang zwischen einer Bedingung und einer Konsequenzen: Ist die Bedingung erfüllt, tritt mit einer gewissen Wahrscheinlichkeit die Konsequenz ein. Die Wahrscheinlichkeit, mit der die Bedingung auf der untersuchten Datenmenge auftritt, ist der Support. Die Wahrscheinlichkeit, mit der im Falle der Erfüllung der Bedingung die Konsequenz eintritt, heißt Confidence. Die Zugpatterns sind Assoziationsregeln sehr ähnlich: Matcht ein Pattern auf eine Stellung (Bedingung), wird mit einer gewissen Wahrscheinlichkeit der vom Pattern vorgeschlagene Zug gespielt (Konsequenz).
Beim Confidence Filtering werden alle Patterns mit zu geringer Confidence verworfen. In [6] wird diese Art des Filtering ebenfalls verwendet.
Beim Support Filtering werden diejenigen Patterns verworfen, die während des Evalua-

tionsvorgangs nicht oft genug gematcht haben[6]. Bei einem Pattern, das auf dem Evaluation Set nicht oft genug gematcht hat, kann man annehmen, dass es nicht allgemein genug ist, um später im Move Guessing von Nutzen zu sein. Es wird möglicherweise im Move Guessing nie eine Stellung auftauchen, in der ein solches Pattern matcht. Support Filtering wird auch in [22] verwendet.
Welche Schwellwerte für die beiden Filtering-Methoden den besten Tradeoff zwischen Vorhersagegenauigkeit des Move Guessers (s. Abschnitt 4.2) und Größe der Patterndatenbank bieten, habe ich nicht detailliert ausprobiert. Die Ergebnisse in Abschnitt 4.8 zeigen aber zumindest, dass das Filtering auf jeden Fall sinnvoll ist.

Mit der Patternevaluation und dem Filtering ist der Aufbau der Patterndatenbank abgeschlossen.

3.6. Move Guessing

Wie bereits erwähnt, bekommt der Move Guesser eine Anfrage zu einer gegebenen Stellung eine Vorhersage von m unterschiedlichen Zügen zu machen. Hierzu geht der Move Guesser die Patterndatenbank durch, prüft welche Patterns matchen und gibt die Züge der m genauesten Patterns zurück.
Die Patterndatenbank ist zum Einen nach Stages sortiert. Das Stage-Feature hat zur Folge, dass z.B. auf ein Brett aus Stage 1 nur Patterns aus Stage 1 matchen können. Sucht man zu einer gegebenen Stellung die matchenden Patterns, kann die Suche also auf die Patterns mit der richtigen Stage reduziert werden. Zum Anderen wird die Patterndatenbank während ihres Aufbaus stets nach den Patternwerten sortiert. Durch diese Sortierung ist es nicht nötig zum Finden der Patterns mit den höchsten Genauigkeitswerten die gesamte Datenbank durchzugehen. Es reicht, wenn man sie beim bestbewerteten Pattern beginnend durchsucht und bei jedem matchenden Pattern den Zug zurück liefert. Sobald man m Züge gefunden hat, kann die Suche abgebrochen werden, da weiter hinten in der Patterndatenbank nur noch Patterns mit niedrigerer Güte gefunden werden könnten.

3.7. Parallelisierbarkeit

Die drei rechenintensiven Aufgaben, die Patterngenerierung, die Patternevaluierung und das Move Guessing, lassen sich ohne großen Kommunikationsaufwand der Prozesse untereinander auf einer Mehrprozessormaschine parallelisieren.
In der Patterngenerierung wird nur lesender Zugriff auf das Training Set und Schreibzugriff auf die (zu generierende) Patterndatenbank benötigt. Um das Programmieren einer Zugriffsverwaltung auf die Patterndatenbank zu vermeiden, arbeiten die Prozesse der Pattergenerierung zunächst unabhängig. Jeder Prozess bekommt eine Teilmenge des Training Sets zugewiesen und sie generieren daraus unabhängig voneinander Patterndatenbanken, die hinterher zu einer Gesamtdatenbank zusammengefügt werden. Beim

[6] Ob sie auch gespielt wurden, ist dabei unerheblich.

Zusammenfügen werden doppelt vorkommende Patterns eliminiert.
In der Patternevaluierung wird ähnlich gearbeitet. Die Patterndatenbank wird aufgeteilt und jeder Prozess evaluiert einen Teil davon. Der hierzu nötige Zugriff auf das Training Set ist nur lesend.
Da der Kommunikationsaufwand verglichen mit der Dauer der sonstigen Berechnungen vernachlässigbar ist, kann man sagen, dass der Lernalgorithmus nahezu ideal parallelisierbar ist.

Das Move Guessing lässt sich ebenfalls parallelisieren. Wie in Abschnitt 3.6 beschrieben ist die Patterndatenbank nach absteigender Güte der Patterns sortiert. Die Prozesse müssen sie nach Patterns, die auf eine gegebene Stellung matchen, durchsuchen. Bspw. bei zwei Prozessen könnte einer die Patterns mit geradem und der andere die mit ungeradem Index durchsuchen. Die Datenbank wird also weiterhin absteigend durchsucht. Wird ein matchendes Pattern gefunden, wird dieses in eine Liste eingetragen. Sind genug Patterns gefunden, wird die Suche abgebrochen.
Auf die Patterndatenbank wird nur lesender Zugriff benötigt. Die Prozesse müssen aber in die selbe Liste der gefundenen Patterns schreiben, da sie sonst nicht wissen, wann genug gefunden sind und die Suche abgebrochen werden kann. Bei der Liste ist also eine Synchronisierung notwendig. Die Parallelisierung dieses Programmteils habe ich nicht realisiert, aber ich vermute, dass auch dieser Teil nur wenig Kommunikationsoverhead hätte und damit (auf einem Zwei-Prozessor-Rechner) auch nahezu doppelt so schnell liefe.

4. Ergebnisse und Schlussfolgerungen

4.1. Verwendete Go-Datenbanken

Da 9x9 Go meist nur zu Lern- und Demonstrationszwecken oder „mal schnell zwischendurch“ gespielt wird, sind nur wenige 9x9 Partien mit sehr hohem Niveau zu finden. Zum Einen habe ich eine Datenbank mit 240 Partien japanischer Profis (im Folgenden GoBase-Set genannt) gefunden. Zum Anderen gab es ein Archiv mit Spielen, die Amateure auf dem No Name Go Server gespielt haben, aus dem ich 558 9x9 Spiele, in denen beide Spieler mindestens 5k waren, herausgefiltert habe (NNGS-Set).
Für 19x19 gibt es sehr viele große Datenbanken auch auf höchstem Niveau, sodass man hier nur auszuwählen brauchte. Die eine Datenbank, die ich gewählt habe, enthält 1265 Spiele der prestigeträchtigsten und höchstdotierten japanischen Turniere[1] (Set 1). Die andere (Set 2) enthält 568 Spiele, ausschließlich von Chô Chikun, der in den 1980er Jahren die japanische Profiwelt dominierte. Beide Datenbanken sind so groß, dass sie in akzeptabler Zeit und mit dem zur Verfügung stehenden Speicher nicht komplett ausgenutzt werden können. Daher habe ich mich in den Tests jeweils nur auf eine Teilmenge aus ihnen beschränkt.

4.2. Definition der Vorhersagegenauigkeit

In manchen Arbeiten (z.B. [6]) wird der Vorhersagealgorithmus dadurch evaluiert, dass man vergleicht, wie sehr die Integration des Algorithmus in einen bestehenden Algorithmus (z.B. UCT mit diversen anderen schon existierenden Erweiterungen) diesen verbessert. Dieses Maß bildet direkt das ab, was von Interesse ist, nämlich die Spielstärke des Gesamtalgorithmus. Es hat jedoch den Nachteil, dass es den Vorhersagealgorithmus nicht isoliert betrachtet. Ob eine Verbesserung direkt aus dem neuen Algorithmus oder aus gutem Zusammenspiel mehrerer Komponenten oder gesunkenem Zeitverbrauch resultiert, kann nicht unmittelbar beurteilt werden.

Daher wird der Algorithmus in vielen Arbeiten isoliert beurteilt ([10, 22, 25]). Seine Aufgabe ist es, zu gegebenen Stellungen jeweils m Züge, die er für gut hält, zurückzuliefern. Es liegt daher nahe, ihm Beispiele aus einem Test Set vorzulegen und zu prüfen, mit welcher relativen Häufigkeit sich der vom Profi gespielte Zug unter den m Vorhersagen befindet.
Wäre UCT fähig unter den vorgeschlagenen Zügen mit Sicherheit den besten auszuwählen, so gäbe diese Wahrscheinlichkeit direkt die Wahrscheinlichkeit an, mit der der

[1] Kisei, Meijin, Honinbô, Jûdan, Tengen, Ôza, Gosei, ...

Gesamtalgorithmus den Profizug spielen würde. Aus diesem Maß lässt sich zwar nicht unmittelbar die Spielstärke des Gesamtalgorithmus ablesen, da zum Einen UCT nicht in der Lage ist, mit Sicherheit den besten Zug aus einer Liste an Zügen auszuwählen und zum Anderen die Fehlerwahrscheinlichkeit alleine keinen Schluss über die Schwere und Auswirkungen der Fehler zulässt. Generell ist dieses Maß aber recht gut geeignet, um die Qualität des Move Guessers zumindest abzuschätzen oder eine Verbesserung im Move Guesser durch Hinzufügen eines neuen Features zu erkennen.

Da die Patterndatenbank an die Grenzen des zur Verfügung stehenden Speichers stößt und auch Zeit ein wichtiges Thema ist, muss zusätzlich zur Vorhersagegenauigkeit auch der Speicherverbrauch und die Geschwindigkeit des Algorithmus beurteilt werden.

4.3. Performanz in 9x9 Go

Für diesen Test wurde der Move Guesser auf 178 Spielen des Gobase-Sets trainiert. Er hat daraus 8263 Trainingsbeispiele extrahiert und 11941 Patterns generiert. Der Grund dafür, dass mehr Patterns erzeugt wurden, als Trainingsbeispiele vorhanden sind, ist, dass aus jedem einzelnen Trainingsbeispiel jeweils Patterns mit 4 bis 9 enthaltenen Steinen erzeugt werden (s. Abschnitt 3.2.1). Die erzeugte Patterndatenbank belegte weniger als 100MB Arbeitsspeicher.
Das verwendete Test Set umfasst 58 Spiele mit 2683 Testbeispielen. Die erzielte Vorhersagegenauigkeit ist in Abbildung 4.1 zu sehen. Auf der x-Achse ist m, die Zahl der Züge, die der Move Guesser vorschlagen darf, aufgetragen. Auf der y-Achse ist die Wahrscheinlichkeit, dass sich der gespielte Profizug unter den m Vorschlägen befindet, aufgetragen. Bspw. ist die Wahrscheinlichkeit, dass sich bei einer Vorhersage von 20 Zügen der gespielte Profizug unter diesen 20 Zügen befindet, ca. 88%. Der Grenzwert der Genauigkeit des Move Guessers liegt bei ca. 92%. Der durchschnittliche Ranking Index (die durchschnittliche Position des Profizugs in den Vorhersagen des Move Guessers) ist ca. 9.
Damit man die Werte des Move Guessers besser vergleichen und interpretieren kann, befindet sich in der Grafik noch je eine Kurve für GnuGo und für einen hypothetischen Move Guesser, der nur zufällige Züge rät. GnuGo wurde hierzu das selbe Test Set vorgelegt, das auch der Move Guesser bearbeiten musste. GnuGo wurde in der Version 3.6 mit Standardeinstellungen (Level 10) eingesetzt. Da GnuGo im Normalfall zu einer gegebenen Stellung nur einen Zug ausgibt, wurde seine Genauigkeit einfach als Konstante aufgetragen. Man hätte es auch anweisen können, mehr als nur einen Zug auszugeben. Allerdings wäre dann die automatisierte Interpretation seiner Ausgaben sehr aufwändig gewesen.
Auffällig ist, dass der Move Guesser bereits für m=3 GnuGo deutlich überholt hat, d.h. wenn UCT zuverlässig aus den 3 ersten Vorhersagen immer den besten herausfinden könnte, würde man vermutlich besser als GnuGo spielen. Ein Aspekt, der ebenfalls nicht unerwähnt bleiben soll, ist die Geschwindigkeit: Für m=3 braucht der Move Guesser zur Abarbeitung des Test Sets 7 Sekunden. GnuGo hingegen braucht über 63 Minuten. Ein Teil dieser Zeit ist zwar auch auf Overhead, der durch die Kommunikation mit GnuGo

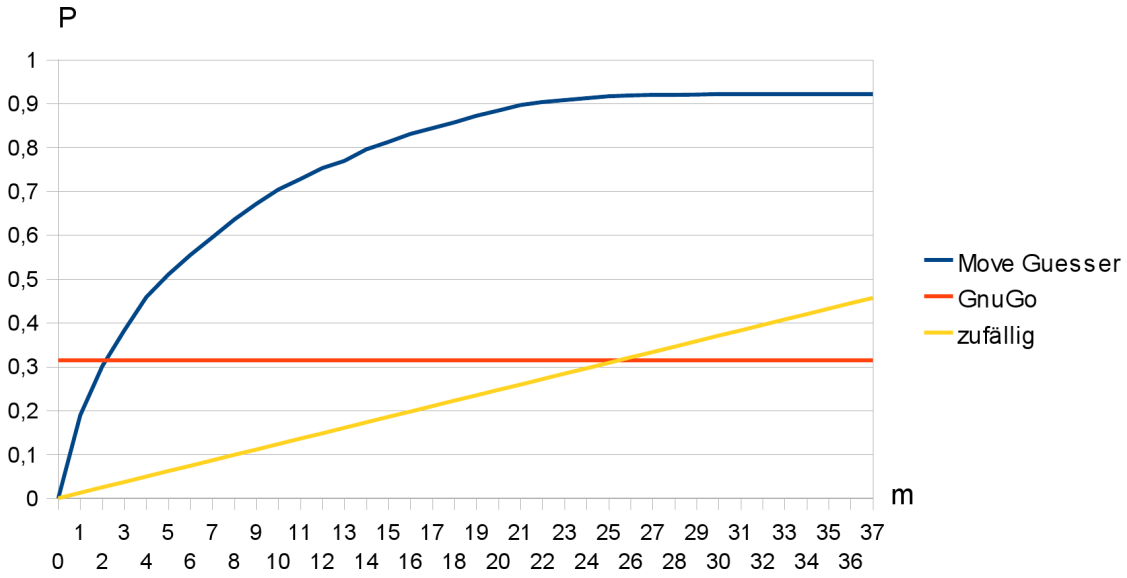


Abbildung 4.1.: Vorhersagegenauigkeit in 9x9

entsteht, zurückzuführen, die Aussage ändert sich dadurch aber nicht wesentlich.

4.4. Performanz in 19x19 Go

Für diesen Test wurden 274 Spiele des Set 1 als Training Set benutzt. Hier tritt allerdings ein Problem auf. Der Speicherbedarf der Patterndatenbank für 19x19 Go ist auf Grund der größeren Bitboards und der hohen Zahl an Translationen eines Patterns sehr groß. Würde man aus allen 274 Spielen Patterns extrahieren, würde der Arbeitsspeicher bei weitem nicht ausreichen und daher ist es nicht möglich viele Spiele zur Patterngenerierung heranzuziehen. Daher wurde nur ein Teil von ihnen, nämlich 28 Spiele, das sog. Acquisition Set, benutzt, um daraus Patterns zu generieren. Anschließend wird wie gewohnt das ganze Training Set zur Patternevaluierung benutzt, um durch ein großes Training Set eine hohe Genauigkeit in der Patternevaluierung zu erreichen.

Es wurden 12007 Patterns generiert und die entstandene Patterndatenbank war ca. 1GB groß. Das Test Set umfasste 18698 Beispiele, ebenfalls aus Set 1 entnommen. Das Ergebnis ist in Abbildung 4.2 zu sehen.

Die Genauigkeit des Move Guessers konvergiert gegen ca. 71%. In 29% der Fälle hat der Move Guesser es also selbst mit beliebig großem m nicht geschafft, den Profizug zu finden. Der durchschnittliche Ranking Index liegt bei 65.8. Hier besteht also durchaus Verbesserungsbedarf.

Zieht man zur Beurteilung wieder GnuGo heran, fällt auf, dass diesmal m=5 nötig ist, um eine höhere Genauigkeit als GnuGo zu erzielen. Auch hier gilt die Aussage, dass der reine Move Guesser wesentlich schneller als GnuGo ist.

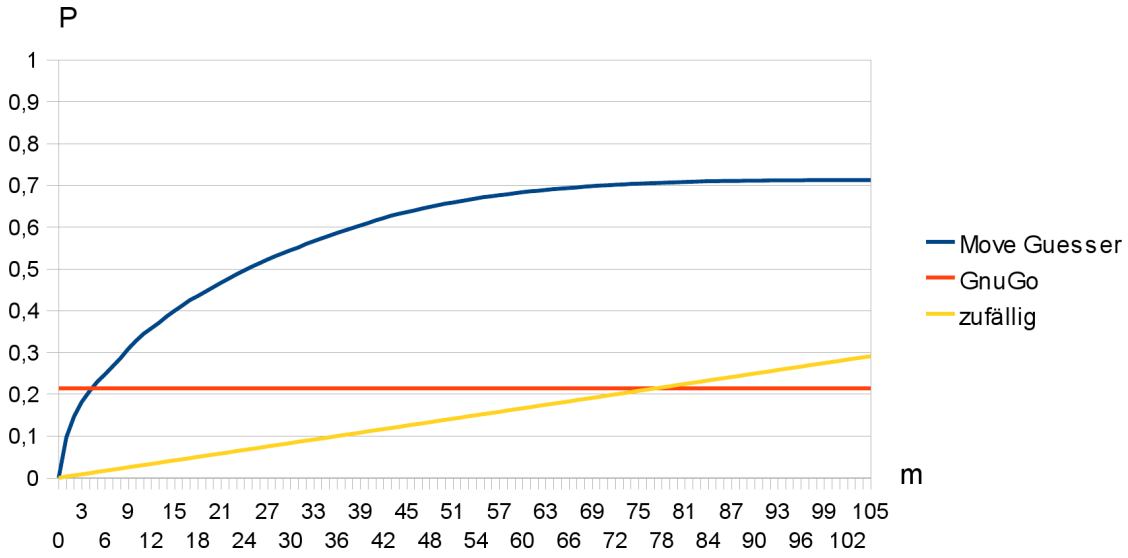


Abbildung 4.2.: Vorhersagegenauigkeit in 19x19

4.5. Performanz in der Eröffnung

Die Eröffnung (insbesondere auf 19x19) ist aus mehreren Gründen von besonderem Interesse. KIs fällt es besonders schwer ohne zusätzliches Wissen die Eröffnung gut zu spielen. Im Schach gibt es zu diesem Zweck Eröffnungsdatenbanken. Wie in Abschnitt 1.1 erwähnt, gibt es diese für Go nicht. Daher soll hier die Genauigkeit des Move Guessers in der Eröffnung evaluiert werden.
Der Move Guesser wurde jeweils auf den ersten 30 Zügen jedes Spiels des kompletten Set 1 trainiert (37950 Trainingsbeispiele). Es wurden 13892 Patterns generiert, die ca. 340MB Arbeitsspeicher einnahmen. Als Test Set wurden jeweils die ersten 30 Züge aller Spiele des Set 2 (35130 Testbeispiele) verwendet. Die erreichte Genauigkeit ist in Abbildung 4.3 zu sehen.

Gleich mit dem ersten Vorschlag (m=1) wird eine sehr hohe Genauigkeit von über 23% erzielt. Bereits mit 2 Vorschlägen ist sie bei über 31% und damit deutlich über GnuGo (25%). Mit größer werdendem m flacht die Zunahme allerdings schnell ab und konvergiert insgesamt gegen ca. 90%. Der Ranking Index liegt bei ca. 29.5.

4.6. Geschwindigkeit des Move Guessers

In 9x9 hat sich gezeigt, dass die Geschwindigkeit des Move Guessers absolut zufriedenstellend ist. Er schafft über 380 Zugvorhersagen pro Sekunde.
In 19x19 hingegen gibt es mehr mögliche Patterns, jedes Pattern hat mehr Translationen, jede Translation braucht größere Bitboards zur Darstellung und daher dauert der Matching-Vorgang und das Durchsuchen der Patterndatenbank insgesamt länger. Der

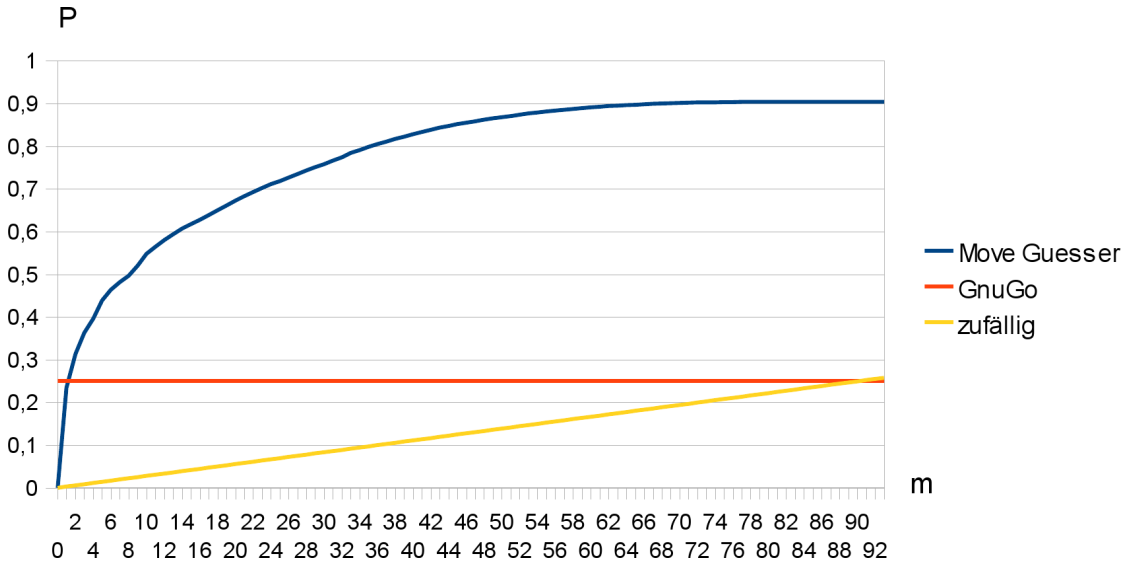


Abbildung 4.3.: Vorhersagegenauigkeit in der Eröffnung

Move Guesser schafft bei m=20 ca. 17 Vorhersagen pro Sekunde.
Nimmt man an, dass eine KI pro Zug eine Minute nachdenken darf und davon die Hälfte der Zeit auf den Move Guesser entfallen darf, könnte er nur knapp über 500 mal befragt werden. Eine Verbesserung um mindestens eine Größenordnung wäre auf jeden Fall wünschenswert.

Eine sicher funktionierende Möglichkeit ist die in Abschnitt 3.7 angesprochene Parallelisierung. Hashing (s. Abschnitt 5.4) birgt ebenfalls ein großes Potential. Eine weitere Möglichkeit wäre, wenn bei jedem Move Guessing ein großer Teil der Patterndatenbank von vornherein verworfen werden könnte. Das geschieht bereits über das Stage-Feature (Patterns aus einer nicht passenden Stage werden vorweg direkt verworfen), aber auch geometrische Datenstrukturen zur Ordnung der Patterns in der Datenbank (s. Abschnitt 5.2) wären denkbar.

4.7. Stage-Feature

In diesem Abschnitt soll gezeigt werden, wie sich das Stage-Feature auf die Performanz (Vorhersagegenauigkeit und Geschwindigkeit) des Move Guessers auswirkt. Hierzu wird der Test zur Performanz in 9x9 Go aus Abschnitt 4.3 wiederholt, einmal mit an- und einmal mit abgeschaltetem Stage-Feature[2].
In Abbildung 4.4 sieht man die Genauigkeit mit und ohne Stage-Feature. Die Genau-

[2]Das Stage-Feature wird nicht wirklich abgeschaltet, sondern die Zahl der Stages lediglich auf 1 gesetzt. Damit verhält sich der Move Guesser so, als ob es das Feature nicht gäbe.

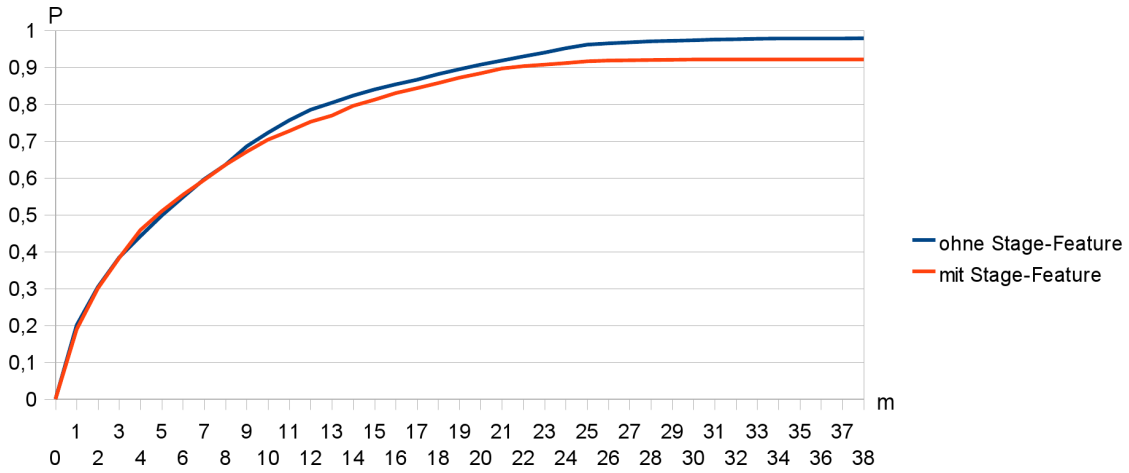


Abbildung 4.4.: Vorhersagegenauigkeit in 9x9 mit und ohne Stage-Feature

igkeit ist mit Stage-Feature etwas niedriger. Dies ist vermutlich hauptsächlich darauf zurückzuführen, dass das Training Set sehr klein ist und durch das Zerteilen des Spiels in mehrere Stages nicht mehr sehr viele Beispiele pro Stage zur Verfügung stehen. Dennoch ist zu sehen, dass die Genauigkeit nicht sehr stark leidet. Für m=20 beispielsweise ist der Unterschied kleiner als 2.5%.
Mit Stages hat die Abarbeitung des Test Sets nur ca. ein Sechstel der Zeit gebraucht. Grundsätzlich kann man sagen, dass das Hinzufügen des Stage-Features absolut durch den Zeitgewinn gerechtfertigt wird.

4.8. Pattern Filtering

Hier wird ebenfalls der Test aus Abschnitt 4.3 wiederholt, diesmal nur mit abgeschaltetem Pattern Filtering (s. Abschnitt 3.5). Hierzu wird keine vergleichende Graphik gezeigt, da die beiden Kurven sich so ähneln, dass nicht viel zu erkennen wäre. Ohne Pattern Filtering ist der Move Guesser nur minimal besser, nämlich bei m=20 um 0.7% und bei m=37 um 0.8%. Mit Filtering wurden 11941 und ohne 30293 Patterns akquiriert. Ohne Filtering hat auf Grund der größeren Patterndatenbank auch die Abarbeitung des Test Sets ein wenig länger gebraucht[3].
Standardmäßig werden Patterns gefiltert, die weniger als 3 Mal auf dem Training Set gematcht oder einen Wert kleiner als 0.01 haben. Für größere Werte reagiert die Genauigkeit des 9x9-Move Guessers relativ empfindlich auf Veränderungen. Ich habe an diesen Werten nicht detailliert herumexperimentiert, nehme aber an, dass wenn Tests mit einem größeren Training Set möglich wären, die Größe der Patterndatenbank über diese

[3]Dass es nur ein wenig und nicht viel länger gedauert hat, ist vermutlich darauf zurückzuführen, dass Treffer in der Patterndatenbank meist schon recht weit oben, bei den Pattern mit hoher Bewertung, geschehen, was auch so beabsichtigt ist.

Konstanten noch weiter gesenkt werden könnte, ohne bedeutende Genauigkeitseinbußen in Kauf nehmen zu müssen.

4.9. Variables k

Wie bereits erwähnt, wird in dieser Arbeit mit einem variablen k für die K-Nearest-Neighbor Representation gearbeitet. Die Auswahl des „richtigen“ k wird dem Pattern Filtering überlassen. Eine Auswertung, wie viele Patterns für jedes k akquiriert wurden, hat das Ergebnis in Abbildung 4.5 ergeben.
Dieses Ergebnis zu interpretieren ist etwas schwierig. Es lässt sich aber zumindest fest-

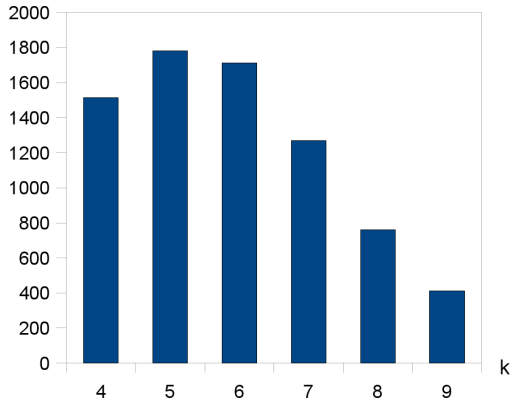


Abbildung 4.5.: Anzahl der Patterns für jedes k

stellen, dass in der Tat Patterns mit unterschiedlichem k erzeugt wurden. Die Kurve hat ihr Maximum bei k=5 und flacht nach rechts und links hin ab.
Dass die Kurve bei k=4 noch nicht sehr flach ist, könnte ein Indiz dafür sein, dass man auch k=3 oder 2 hätte zulassen müssen. Allerdings enthalten erfahrungsgemäß zumindest Patterns mit 2 Steinen zu wenige Informationen, um relevant zu sein. Daher ist das Aussehen der Kurve an dieser Stelle evtl. auch einfach nur darauf zurückzuführen, dass im Confidence Filtering der Confidence Treshold nicht hoch genug gesetzt war und dadurch nicht streng genug zu allgemeine Patterns weggefiltert wurden.

4.10. Speicherverbrauch

Auf 9x9 hat sich gezeigt, dass man mit dem momentanen Speicherverbrauch auch auf wesentlich größeren Test Sets hätte arbeiten können. Leider habe ich keine andere große Datenbank mit Profispielen finden können. Wie bereits erwähnt, kann man sich mit Spielen, die auf einem Go-Server mitprotokolliert wurden, aushelfen. Hieraus ließen sich allerdings auch nicht viele Spiele mit ausreichendem Niveau extrahieren.
Es wäre wünschenswert gewesen, eine mindestens zehnmal so große Datenbank zu haben. Dann hätten auch zusätzlich zu den bisher schon vorhandenen Features noch weitere

hinzugenommen werden können, mit denen dann noch spezialisiertere Patterns hätten gelernt werden können. Mit der aktuellen Datenbankgröße hätte es allerdings keinen Sinn gehabt, die Patterns weiter zu spezialisieren, da auch jetzt schon in den Ergebnissen erkennbar ist, dass von der Patterndatenbank nicht jede Situation abgedeckt wird.

Auf 19x19 ergibt sich das Problem, dass die Patterndatenbank zu viel Speicher belebt. Es können nicht sehr viele Patterns akquiriert werden, da die Patterns auf Grund der größeren Bitboards und der größeren Zahl an möglichen Translationen wesentlich mehr Platz verbrauchen. Daher wurde in den Tests nur mit einer Teilmenge der zur Verfügung stehenden Spieldatenbanken gearbeitet, sodass der von der Patterndatenbank in Anspruch genommene Speicher gerade so an die von Java gegebene Beschränkung (ca 1.6 GB) heranreicht.
Wie auch auf 9x9 wäre es wünschenswert, mehr Patterns mit zusätzlichen Features akquirieren zu können. Einen Ansatz, wie dies dennoch realisiert werden könnte, findet man in Abschnitt 5.1.

5. Denkbare Erweiterungen und Anwendungsmöglichkeiten

5.1. Spezialisierte Move Guesser für einzelne Stages

Eine weitere Möglichkeit, um der Größe der Patterndatenbank Herr zu werden, ist das Trainieren von Move Guessern, die auf einzelne Stages des Spiels spezialisiert sind. Die Ergebnisse in Abschnitt 4.7 haben gezeigt, dass die Einführung des Stage-Features nur eine kleine Verschlechterungen in der Vorhersagequalität gebracht hat. Man kann also relativ „gefahrlos“ Move Guesser trainieren, die ausschließlich auf eine einzelne Stage spezialisiert sind.
Dies geschieht, indem man dem Move Guesser der jeweiligen Stage nur Trainingsbeispiele aus dieser Stage vorlegt. Für jede Stage kann man einen Move Guesser trainieren und ihn samt zugehöriger Patterndatenbanken auf der Festplatte ablegen. Zu Anfang des Spiels wird dann der erste Move Guesser von der Platte geladen und mit jedem Wechsel der Stage wird der alte Move Guesser aus dem Arbeitsspeicher entfernt und der neue nachgeladen. Damit befindet sich jeweils nur eine wesentlich kleinere Patterndatenbank im Speicher.
Diese Platzersparnis kann z.B. dazu genutzt werden, um auf einem größeren Training Set zu lernen, damit die Vorhersagequalität zu erhöhen und evtl. noch weitere Features in die Patterns aufzunehmen.

5.2. Räumliche Datenstruktur

In [4] wurde die Verwendung einer geometrischen Datenstruktur zur Organisation der Patterns in der Patterndatenbank vorgeschlagen. Beispielsweise könnten die Patterns in einem Quadtree abgelegt werden. Versucht man dann ein zu einem Brett matchendes Pattern zu finden, braucht man jeweils nur im in Frage kommenden Teilbaum suchen.
Man hat bei diesem Ansatz allerdings mit der Existenz von „Don't Cares“, also Feldern, deren Inhalt beliebig ist, zu kämpfen[1]. In meiner Arbeit arbeite ich auch mit „Don't Cares“. Es wird sogar nichtmal zwischen leeren Feldern und „Don't Cares“ unterschieden (vgl. Abschnitt 3.2.1). Verwendet man „Don't Cares“, so ist nicht klar, in welchem Teilbaum ein Pattern exakt abzulegen ist. In obiger Diskussion wird vorgeschlagen jeden möglichen Wert (also schwarz, weiß und leer) in das „Don't Care“ einzusetzen und die Duplikate allesamt in der Datenstruktur abzulegen. Mit der Anzahl der „Don't Cares“ wächst die Zahl der Duplikate exponentiell. Angesichts dessen, dass die in dieser Arbeit

[1] vgl. auch `http://computer-go.org/pipermail/computer-go/2006-June/005590.html`

verwendeten Patterns sehr viele „Don't Cares“ enthalten, ist dieses Vorgehen absolut nicht praktikabel.
Diese Idee wurde hier dennoch erwähnt, da es möglicherweise noch einen anderen Weg gibt, die Patterns geeignet zu organisieren, um den Matching-Vorgang zu beschleunigen.

5.3. Dependency Set

In [19] werden sog. Dependency Sets vorgeschlagen: Jedes Feld des Bretts wird mit den Patterns, die von diesem Feld abhängig sind, assoziiert. Dadurch braucht nur einmal ganz zu Anfang eines Spiels berechnet werden, welche Patterns auf das Brett matchen und welche nicht. Im Verlauf des Spiels werden dann Züge auf dem Brett ausgeführt und der Matching-Status der Patterns braucht nur für jene Patterns neu berechnet werden, die von einem oder mehreren der durch die Züge veränderten Felder abhängen. [19] zufolge können dadurch in realen Spielen mind. 93% Rechenzeit eingespart werden.

5.4. Hashfunktion

Eine Erweiterung der Bitboards, die ich gerne realisiert hätte, wofür ich aber leider keine Lösung gefunden habe, arbeitet mit Hashfunktionen. Die Idee wird durch die Verwendung von Zobrist-Hashing (s. [28]) in Schach-KIs motiviert. Sie basiert darauf, die Bitboards nicht in ihrer normalen Version zu speichern, sondern auf sie eine Hashfunktion anzuwenden und die Hashwerte zu speichern. Die Anforderung an die Hashfunktion ist, dass sie den Sinn der binären and-Operation erhalten muss.
Die Hashfunktion h muss ein Homomorphismus zwischen den Bitboards mit der normalen and-Operation und den gehashten Bitboards mit einer and*-Operation sein. Es muss also eine Hashfunktion h und eine and*-Operation &* geben sodass gilt: h(Brett-Bitboard) &* h(Maske) = h(Brett-Bitboard & Maske). Wünschenswert ist natürlich, dass &* ebenfalls möglichst einfach auf einem Prozessor ausführbar ist, z.B. indem es identisch zu & ist. Außerdem sollte h schnell berechenbar sein. h(Maske) sowie h(Pattern-Bitboard) sind nicht zeitkritisch, da sie nur einmal pro Pattern ausgeführt werden müssen. h(Brett-Bitboard) hingegen ist sowohl im Lernvorgang als auch im Move Guessing zeitkritisch.
Außerdem sollte h möglichst kollisionsresistent sein, es sollte also unwahrscheinlich sein, dass man ein Paar von Bitboards b und b* findet, sodass h(b)=h(b*) gilt. Für Zobrist-Hashing ist eine Kollision, wenn man die Zobrist-Keys 64 Bit breit wählt, erst ab ca. 2^{32} Bitboards wahrscheinlich.[2]
Wenn diese Erweiterung mit Zobrist-Hashing möglich gewesen wäre, wäre h(Brett-Bitboard) schnell berechenbar gewesen, da die Zobrist-Hashwerte im Laufe des Spiels inkrementell geupdatet werden können. Man beginnt zu Beginn des Spiels mit dem Hashwert der Startkonfiguration (z.B. h(leeres Brett)) und kann bei jedem Hinzufügen oder Ent-

[2] Auf Grund des Geburtstagsparadoxons gilt nämlich: $P(\text{Kollision}) = \frac{(n-1)\cdot n}{2} \cdot 2^{-64} > 0.5 \Rightarrow n \approx 2^{32}$

fernen eines Steins den Hashwert inkrementell updaten.[3] Leider erfüllt Zobrist-Hashing (mit &* identisch zu &) nicht das oben genannte Kriterium und ich habe noch keine Alternative gefunden.
Das Hashing hätte den Vorteil gehabt, dass die Patterns (mit 64 Bit breiten Zobrist-Keys) nur noch ein Sechstel des Platzes und das Matching nur noch ein Sechstel der Zeit beansprucht hätten (selbstverständlich mit dem oben erwähnten Trade-Off, dass nun zusätzlich die Hashwerte der Bretter berechnet oder immer geupdatet werden müssten).
Eine weitere sehr praktische Eigenschaft einer Hashfunktion wäre es, wenn sich aus dem Hashwert eines Bretts schnell die Brettrotationen und -translationen berechnen ließen. Dann müssten Sie nicht mehr beim Lernen vorherberechnet und abgespeichert werden, sondern könnten live beim Matching berechnet werden.

5.5. Massive Parallelisierung

In Abschnitt 3.7 wurde bereits das Thema Parallelisierung angesprochen. Insbesondere ist hier die Parallelisierung des Move Guessers von Interesse, da nur der Move Guesser bei einer Anwendung in einer Go-KI zeitkritisch wäre.
Der Algorithmus ist gut parallelisierbar (vgl. Abschnitt 3.7) und würde sich daher zur Implementierung auf einem massiv-parallelen System eignen. Durch GPGPU[4] und Programmiersprachen wie CUDA oder OpenCL sind massiv-parallele Systeme erschwinglich und die Programmierung relativ einfach machbar geworden. Der Implementierung auf einer Grafikkarte käme auch zu Gute, dass das Move Guessing nur aus einer linearen Suche durch die Pattern-Datenbank besteht und die Operationen zum Matching einfache binäre Operationen sind.
Eine massiv-parallele Implementierung könnte sich also lohnen, zumal auch Beschleunigungen des Move Guessers um Faktor 10 bis 100 durchaus denkbar sind.

5.6. Anwendungsmöglichkeiten des Algorithmus

Der Move Guesser alleine ist nicht brauchbar, um Computer-Go auf hohem Niveau zu spielen. Bspw. lernt er nicht, wie er auf schlechte Züge und Fehler seines Gegners reagieren soll, da in den Profispielen keine schlechten Züge vorkommen und er nie in einem Beispiel sieht, wie er solche Fehler bestrafen soll. Außerdem gibt es viele andere Konzepte des Spiels, wie z.B. Interferenz zwischen unterschiedlichen Bereichen des Brettes, die der Move Guesser auf Grund seiner eher einfach gehaltenen Patterns nicht zu fassen vermag. Das größte Problem aber ist, dass das Move Guessing höchstens eine Bewertung der momentanen Situation darstellt und in keinster Weise die Stellung in die Zukunft untersucht.[5]

[3] Diese praktische Eigenschaft macht Zobrist-Hashing auch in der Spiele-KI-Programmierung, z.B. für Transposition Tables im Schach oder für Super-Ko-Detection im Go, so populär.
[4] General Purpose Computation on Graphics Processing Unit
[5] Dieses Problem wurde bereits in Abschnitt 2.1 im Zusammenhang mit Neuronalen Netzen angesprochen.

Es ist daher nötig, dass man die Zugvorschläge durch einen weiteren Algorithmus, der ihre Konsequenzen in die Zukunft untersucht, verifizieren und einen der Vorschläge aussuchen lässt. Dies kann durch einen Spielbaum-Suchalgorithmus geschehen.
Betrachtet man die Kombination des Move Guessers mit einem Spielbaum-Suchalgorithmus von der Seite des Suchalgorithmus aus, erscheint die Kombination ebenfalls sinnvoll: Wie bereits in Abschnitt 1.1 erwähnt, hat man in Go stets mit dem hohen Verzweigungsfaktor zu kämpfen. Der Move Guesser ist dazu geeignet den Verzweigungsfaktor zu verkleinern, indem man nur die von ihm gemachten Vorschläge untersucht. Er kann also z.B. in Alpha-Beta als Zuggenerator benutzt werden, der an einem Knoten eine festgelegte Anzahl Folgeknoten expandiert.
Eine Implementierung könnte folgendermaßen aussehen[6]:

```
int alphaBeta(int depth, int alpha, int beta) {
  if (depth == 0) return evaluate();
  Vector<Move> followUpMoves = moveGuesser.getMoves(m,
      currentState);
  for (Move move : followUpMoves) {
    currentState.executeMove(move);
    int value = -alphaBeta(depth-1,-beta,-alpha);
    currentState.undoLastMove();
    if (value >= beta) return beta;
    if (value > alpha) alpha = value;
  }
  return alpha;
}
```

Die Konstante m stellt dabei den fest eingestellten Verzweigungsfaktor dar. Der Verzweigungsfaktor lässt sich so beliebig festlegen. Allerdings muss man bei kleineren Verzweigungsfaktoren, da die Vorhersagen des Move Guessers mit Unsicherheiten behaftet sind, einen größeren Genauigkeitsverlust von Alpha-Beta in Kauf nehmen.
Zusätzlich zum Abschneiden von Ästen kann man den Move Guesser auch zum Move Ordering einsetzen. Die Effizienz von Alpha-Beta hängt empfindlich davon ab, wie die Kinder eines Knotens sortiert sind. Mit Hilfe der Vorhersagen des Move Guessers kann man die Kindknoten nach ihrem geschätzten Wert sortieren:

```
int alphaBeta(int depth, int alpha, int beta) {
  if (depth == 0) return evaluate();
  Vector<MoveWithValue> followUpMoves = moveGuesser.
      getMovesWithValues(m, currentState);
  sortByValue(followUpMoves);
  for(MoveWithValue move : followUpMoves) {
    currentState.executeMove(move);
    int value = -alphaBeta(depth-1, -beta, -alpha);
```

[6] Der Code orientiert sich am Beispielcode des deutschsprachigen Wikipedia-Artikel zum Alpha-Beta-Algorithmus.

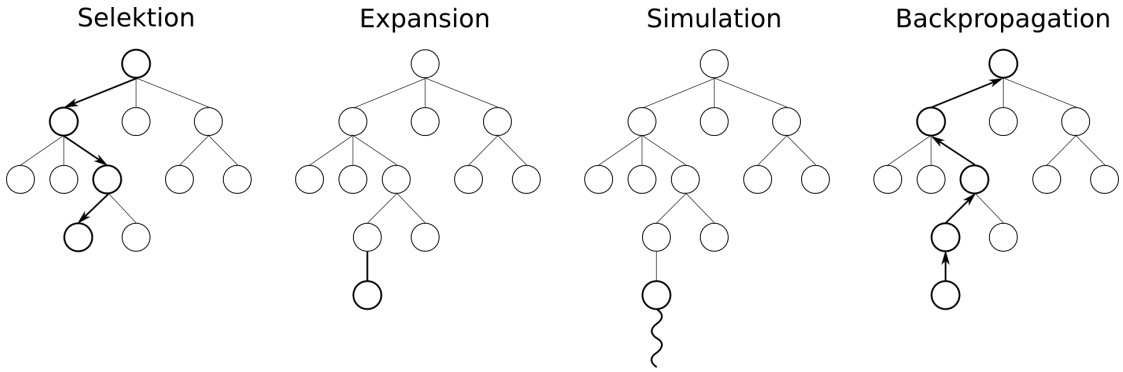


Abbildung 5.1.: Die vier wesentlichen Teile von UCT. Die Zeichnung ist der Zeichnung in [8] nachempfunden.

```
      currentState.undoLastMove();
      if (value >= beta) return beta;
      if (value > alpha) alpha = value;
    }
    return alpha;
}
```

Der Move Guesser soll nun statt Zügen Paare aus Zügen und zugehörigem abgeschätztem Wert liefern. Die Zugliste wird anschließend nach den Wertabschätzungen sortiert und die Züge der Reihe nach durch Alpha-Beta rekursiv untersucht. Je besser die Reihenfolge der Züge ist, desto früher findet ein Cutoff (**return** beta;) statt.

5.6.1. Einsatz in UCT

In den letzten Jahren hat sich ein rechenintensiver Algorithmus, UCT, im Computer-Go etabliert. Neben seiner generellen Eignung für Go[7] ist einer seiner Vorzüge die leichte Integrierbarkeit unterschiedlichster Heuristiken, z.B. auf Patterns basierende Heuristiken. Arbeiten zur Verwendung von Patterns in UCT (bzw. allgemein Monte-Carlo-Algorithmen) sind [6, 10, 15]. Da UCT kein wesentliches Thema dieser Arbeit ist, wird der Algorithmus hier nur angerissen. Eine detailliertere Erklärung findet man in [8].

Der Algorithmus führt mehrere Iterationen von jeweils vier Schritten (s. Abb. 5.1) aus. Im ersten Schritt wählt er von der Wurzel ausgehend einen Weg zu einem Blatt, das er gerne weiter untersuchen würde. Im zweiten Schritt expandiert er einen oder mehrere Kindknoten dieses Blattes. Im dritten Schritt spielt er auf den expandierten Kindknoten ein zufälliges Spiel (Monte-Carlo-Simulation) bis zu einer Terminalstellung[8]. Im vierten Schritt propagiert er die durch die Simulation erhaltene Wertabschätzung des Knotens hoch zu den Elternknoten, bis hin zur Wurzel.

[7] Es wird keine Blatt-Evaluationsfunktion und auch sonst fast kein Wissen über Go benötigt.
[8] eine Stellung, in der das Spiel vorbei ist

Ähnlich wie in Alpha-Beta kann der Move Guesser hier benutzt werden, um den Verzweigungsfaktor zu begrenzen oder um Move Ordering durchzuführen. Die Werte von Kindknoten werden mit dem Move Guesser abgeschätzt und die Kinder damit sortiert. Eine weitere Einsatzmöglichkeit des Move Guessers sind die Monte-Carlo-Simulationen an den Blättern des UCT-Baums. Reines UCT simuliert nur zufällige Go-Spiele bis zu einer Terminalstellung. Mit Hilfe des Move Guessers bräuchte man nicht mehr rein zufällige Züge zu spielen, sondern könnte die von ihm vorgeschlagenen Züge benutzen. In einer konkreten Implementierung könnte UCT folgendermaßen aussehen:

```
Move uct(int numIterations){
  for(int i=0;i<numIterations;i++){
    Node selected=select(root);
    expand(selected);
    double value=simulate(selected.getBestChild());
    backpropagate(value,selected);
  }
  return root.getBestChild();
}

void expand(Node node){
  Vector<MoveWithValue> followUpMoves=moveGuesser.
      getMovesWithValues(m,node);
  sortByValues(followUpMoves);
  for(MoveWithValue move : followUpMoves)
    node.addChild(move,move.value);
}

double simulate(Node node){
  if(isTerminal(node))
    return evaluate(node);
  else{
    Move move=moveGuesser.getMoves(1,node);
    return simulate(node.executeMove(move));
  }
}
```

select (root) selektiert wie bereits erwähnt von der Wurzel ausgehend einen Pfad durch den Baum bis zu einem Blatt. Auf das genaue Aussehen der Funktion wird hier nicht genauer eingegangen. expand(selected) expandiert die Kinder des selektierten Blatts. Der Move Guesser liefert hier nicht nur die möglichen Züge, sondern auch Abschätzungen ihrer Werte (z.B. die Vorhersagegenauigkeit des zum Zug gehörigen Patterns). Die Kinder werden anschließend nach diesen Werten sortiert.
simulate(selected.getBestChild()) führt eine Simulation auf dem besten Kind aus. Ohne das zuvor in der expand-Funktion geschehene Move Ordering wäre es gar nicht feststellbar gewesen, welches das beste Kind ist.

In den Simulationen wird nun ebenfalls der Move Guesser verwendet. Er schlägt in jeder Stellung jeweils nur den vielversprechendsten Zug vor und dieser wird weiter simuliert. An beiden Stellen, in denen im obigen Code der Move Guesser verwendet wird, müsste ohne ihn einfach irgendein zufälliger Knoten ausgewählt werden. Da der Move Guesser hier im Schnitt sicher eine wesentlich sinnvollere Auswahl trifft, stellt seine Verwendung also höchstwahrscheinlich einen qualitativen Gewinn in der Genauigkeit von UCT dar.

Ein anderer Aspekt, der hier ebenfalls beachtet werden muss, ist die Geschwindigkeitseinbuße durch den Move Guesser: Er ist erheblich langsamer als eine rein zufällige Zugauswahl. In [8, Abschnitt Using Time-Expensive Heuristics] wird vorgeschlagen zeitintensive Heuristiken (wie in unserem Fall den Move Guesser) nur in wichtigen Knoten einzusetzen. Wichtig ist ein Knoten dann, wenn er in allen vorherigen Iterationen mindestens so oft wie ein bestimmter Schwellwert (10 in [8]) besucht wurde. Dies könnte folgendermaßen aussehen:

```
void expand(Node node){
  if(node.getNumPreviousIterations()<threshold)
    node.expandRandomChild();
  else if(node.getNumPreviousIterations()==threshold){
        node.throwChildrenAway();
    Vector<MoveWithValue> moves=moveGuesser.getMovesWithValues(
        m,node);
    for(MoveWithValue move : moves)
      node.addChild(move,move.value);
    node.sortChildrenByValue();
  }
}
```

Sind noch wenig Iterationen durch den Knoten gelaufen (node.getNumPreviousIterations ()<threshold), wird jeweils nur ein Kind expandiert (und simuliert). Wird der Knoten oft genug simuliert und hat sich somit als wichtig für das Gesamtergebnis von UCT erwiesen, wird der Move Guesser um eine Bewertung gebeten. Vorher werden die bisherigen Werte der Kinder weggeworfen, was aber kein großer Verlust ist, da die Werte auf Basis der bisherigen Anzahl an Simulationen noch nicht sehr akkurat waren.
Durch die Verwendung des Schwellwerts ist in [8] der zusätzliche Zeitverbrauch des Gesamtalgorithmus nur minimal (um 4%) gestiegen. Ohne seine Verwendung wäre der Algorithmus ca. 1000fach langsamer gewesen.
Genau wie in der expand-Funktion muss auch über die Geschwindigkeit des Move Guessers in der simulate-Funktion nachgedacht werden. Hier verliert man durch den Move Guesser nicht ganz so viel Geschwindigkeit, da der Move Guesser immer nur genau einen Zug zurückliefern muss und er seine Suche nach dem besten Zug in der Pattern-Datenbank abbrechen kann, sobald er das erste passende Pattern gefunden hat. Die Geschwindigkeitseinbuße dürfte dennoch recht groß sein und man müsste in einem Experiment ermitteln, ob diese durch den Genauigkeitsgewinn gerechtfertigt wird.
Eine Möglichkeit der Optimierung besteht allerdings auch hier: Hier sind keine sehr

genauen Vorhersagen des Move Guessers nötig, da selbst ein relativ schlechter Move Guesser besser ist, als eine rein zufällige Zugauswahl. Man könnte hierfür speziell einen leichtgewichtigeren Move Guesser mit einer kleineren Patterndatenbank trainieren, der durch die kleinere Patterndatenbank schnellere Zugvorhersagen machen kann.

5.7. Fazit

Die Abschnitte 4.3 und 4.4 haben gezeigt, dass der Move Guesser sehr erfolgversprechende Resultate in der Zugvorhersage liefert. Darf er 5 Züge vorschlagen, liefert er genauere Resultate als GnuGo mit einem Vorschlag. Auf 9x9 braucht er hierzu sogar nur 3 Vorschläge.
Dieses Ergebnis ist in sofern beachtlich, als dass das Pattern Matching wesentlich einfacher funktioniert, als die komplizierten Berechnungen GnuGos. Außerdem war hierzu, wie es in Abschnitt 1.2 gefordert wurde, keine aufwändige Handkodierung von Expertenwissen nötig. Die Patterndatenbank wird automatisch erzeugt und ihre Qualität (Vollständigkeit und Korrektheit) automatisch sichergestellt.
Auch kann festgestellt werden, dass es gelungen ist, eine gute „Eröffnungsdatenbank“ aufzubauen. In der Eröffnung werden nur 2 Vorschläge gebraucht, um GnuGo zu übertreffen.

Der Move Guesser arbeitet auf Grund der Bitboards sehr schnell. Auf 9x9 schafft er mehr als 380 Vorhersagen pro Sekunde, auf 19x19 immerhin 17.
Ein Problem, das noch in den Griff gebracht werden muss, ist der Speicherverbrauch auf 19x19. Spezialisierte Move Guesser für die einzelnen Stages (Abschnitt 5.1) könnten hier Abhilfe schaffen.

In Zukunft wäre natürlich eine Untersuchung des Einsatzes in UCT interessant, da der Move Guesser seinen Sinn erst dadurch erhält. Ob der Move Guesser hier wie erhofft von Nutzen sein kann, müsste erprobt werden.

A. verwendete Hilfsmittel

A.1. Software

Diese Bachelorarbeit wurde in Java 6 mit Eclipse 3.4 als Entwicklungsumgebung programmiert.
Für die Vergleiche des Move Guessers mit GnuGo in Kapitel 4 wurde GnuGo 3.6 eingesetzt.
Der schriftliche Teil dieser Arbeit wurde mit LaTeX verfasst. Verwendet wurde die LaTeX-Distribution MiKTex 2.7.
Die Graphiken in der Auswertung wurden mit OpenOffice.org und die Bilder von Go-Stellungen mit CGoban 3 erstellt.

A.2. Go-Datenbank

Die für 9x9 Go verwendeten Spiele stammen von `http://computer-go.org/pipermail/computer-go/2006-April/005343.html` sowie `http://www.gobase.org/9x9/`
Die 19x19 Spielesammlungen findet man unter `http://homepages.cwi.nl/~aeb/go/games/index.html` und `http://www.xs4all.nl/~rongen17/Cho/Site/`

B. Die Regeln von Go

Die Go-Regeln zu kennen, ist zum Verständnis dieser Arbeit nicht zwingend notwendig, aber evtl. hilfreich. Sie sind, verglichen mit der Komplexität des Spiels, die daraus erwächst, verhältnismäßig einfach.

Im Go wird nicht wie im Schach in die Felder des Bretts, sondern auf die Kreuzungspunkte der Linien gespielt. Es wird auch auf den äußersten Rändern und Ecken gespielt. Ein normales Brett hat 19x19 Kreuzungspunkte, für Anfänger und zu Übungszwecken werden aber auch 9x9- und 13x13-Bretter verwendet.

B.1. Setzen

Schwarz beginnt stets das Spiel. Beide Spieler legen abwechselnd Steine auf das Brett. Liegen gleichfarbige Steine direkt nebeneinander, bilden sie gemeinsam eine Gruppe. Diagonal benachbarte Steine bilden keine gemeinsame Gruppe. In Abbildung B.1 sieht man ein 9x9-Brett nach sieben Zügen. Die vier mit A markierten schwarzen Steine hängen alle zusammen. Von den drei weißen Steinen sind nur zwei (B) benachbart. Der dritte Stein (C) ist nur diagonal benachbart und bildet alleine eine Gruppe aus nur einem Stein.

B.2. Schlagen

Jede Gruppe hat um sich herum freie Felder, sog. Freiheiten. Jedes leere Feld, das an einen Stein der Gruppe angrenzt, zählt dabei als Freiheit. Freiheiten können sich also auch im Innern einer Gruppe befinden. Schafft es ein Spieler die letzte verbliebene Freiheit einer gegnerischen Gruppe zu belegen, ist diese Gruppe geschlagen, wird vom Feld genommen und zu den Gefangenen gelegt. In Abbildung B.2 kann Weiß mit Spielen auf A, B oder C jeweils eine schwarze Gruppe schlagen. Die Ergebnisse sind im zweiten Bild zu sehen.

B.3. Selbstmordverbot

Es ist nicht erlaubt einen Stein auf die letzte Freiheit einer eigenen Gruppe zu spielen, also Selbstmord zu begehen. In Abbildung B.3 darf Schwarz nicht auf A spielen, da A seine letzte Freiheit ist. Auf B hingegen darf er spielen, da er dadurch die vier weißen Steine schlägt und hinterher wieder vier (in der Abbildung durchnummerierte) Freiheiten hat. C ist aus dem selben Grund erlaubt und der einzelne Stein hat hinterher zwei Freiheiten.

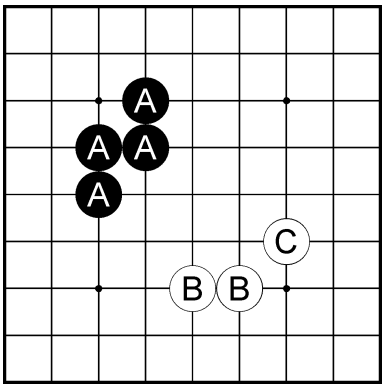


Abbildung B.1.: 3 Gruppen

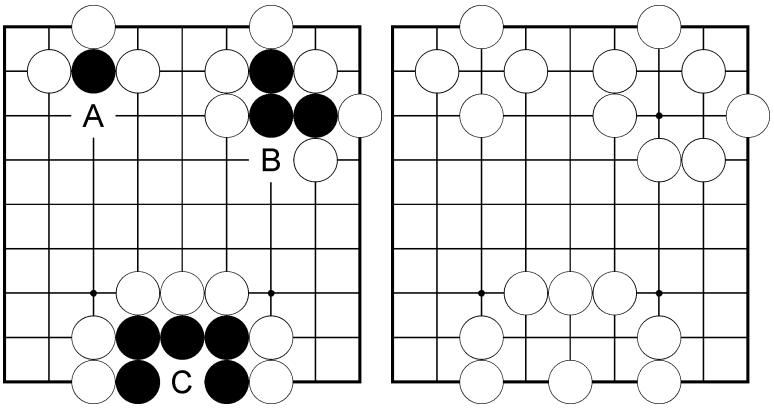


Abbildung B.2.: vor und nach dem Schlagen

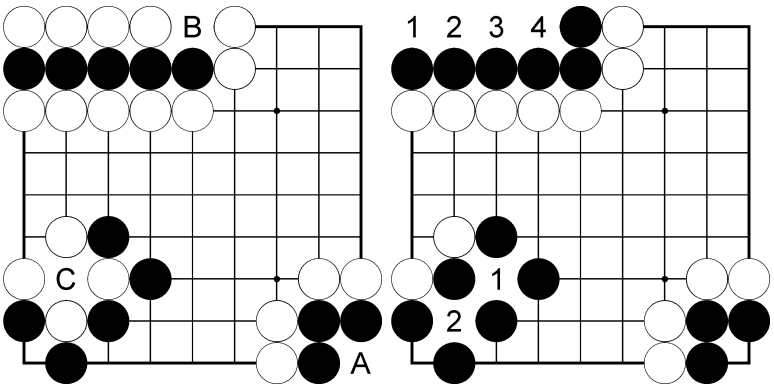


Abbildung B.3.: vor und nach dem Schlagen

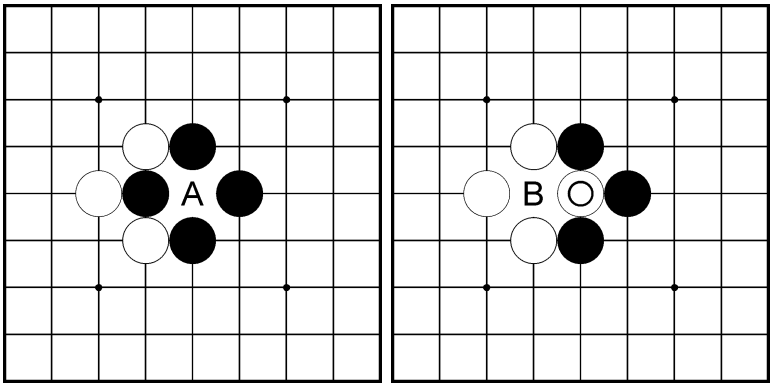


Abbildung B.4.: Ko-Stellung

B.4. Ko-Regel

Häufig kommen Stellungen vor, in denen sich die aktuelle Stellung endlos wiederholen könnte. In Abbildung B.4 sieht man: Weiß kann auf A einen schwarzen Stein schlagen, danach könnte Schwarz auf B direkt zurückschlagen und die Ausgangssituation wäre wieder erreicht. Um nicht in einer Endlosschleife zu landen, müsste einer von beiden einen anderen Zug spielen, woran aber evtl. keiner von beiden interessiert ist. Solche sog. Ko[1]-Stellungen werden durch die Regeln gesondert berücksichtigt. Wird ein einzelner Stein geschlagen, darf der schlagende Stein nicht direkt einzeln zurückgeschlagen werden.
Dies impliziert, dass die Regel keine Anwendung findet, wenn im ersten oder zweiten Zug der Ko-Stellung mehr als ein Stein geschlagen wird, da nach dem Schlagen mehrerer Steine Endlosschleifen nicht mehr ohne weiteres möglich sind[2]. Wichtig ist auch, dass der Stein zwar nicht direkt zurückgeschlagen werden darf, sehr wohl aber nachdem Schwarz irgendwo anders auf dem Brett gespielt hat. Spielt Schwarz in diesem Beispiel an anderer Stelle, kann Weiß auf diese Drohung reagieren und Schwarz kann in der Ko-Stellung zurückschlagen. Weiß kann die Drohung aber auch ignorieren und den einzelnen Stein mit einem Zug auf B mit den anderen dreien verbinden.

B.5. Spielende und Auszählung

Sieht man keine Möglichkeit mehr mit irgendeinem Zug die eigene Stellung zu verbessern, passt man. Passen beide Spieler direkt hintereinander, ist das Spiel vorbei. Punkte bekommt man nun sowohl für freie Gebiete des Bretts, die ausschließlich von eigenen Steinen umschlossen sind, als auch für Steine des Gegners, die man im Laufe des Spiels gefangen hat.

[1] sprich „Koh“

[2] Sie sind zwar noch möglich, aber zumindest die japanischen Regeln verhindern dies nicht. Tritt eine solche Situation auf, kann das Spiel mit beiderseitigem Einverständnis als Unentschieden gewertet werden.

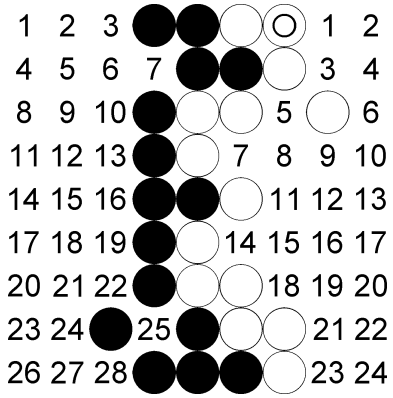


Abbildung B.5.: Eine Endstellung

Zusätzlich bekommt Weiß einen Ausgleich (das sog. Komi) dafür, dass Schwarz den ersten Zug des Spiels hatte. Auf 9x9 sind das z.B. 5.5 und auf 19x19 6.5 Punkte[3]. Der halbe Punkt dient dazu ein Unentschieden zu verhindern. In Abbildung B.5[4] sieht man eine Endstellung eines Spiels. Schwarz hat 28 Gebietspunkte und 1 Gefangenen. Weiß hat 24 Gebietspunkte und 5.5 Punkte Komi. Weiß gewinnt also mit einem halben Punkt Vorsprung.

B.6. Beispiel

Um die Regeln einmal im Einsatz zu sehen, wird hier ein Probespiel auf 9x9 zwischen Go Seigen und Miyamoto Naoki von 1968 gezeigt. Go Seigen galt in seiner Zeit als Go-Wunderkind und wird mittlerweile von vielen als bester Spieler des 20. Jahrhunderts und einer der besten aller Zeiten betrachtet.

Schwarz eröffnet in Abbildung B.6 das Spiel mit dem 4-5-Punkt (4. Spalte von links, 5. Zeile von oben). Weiß antwortet mit 6-5. Nach den Zügen 3 bis 5 sind die Steine 1 und 3, sowie 2 und 4 verbunden. 5 bildet eine Gruppe aus nur einem Stein.
Nach den Zügen 6 bis 8 droht der Stein A geschlagen zu werden.

Hätte Schwarz auf dem Brett etwas Wichtigeres zu tun, könnte er A auch opfern und Weiß überlassen. Er entscheidet sich aber dafür A zu retten und verbindet ihn, wie in Abbildung B.7 zu sehen, mit den anderen zwei schwarzen Steinen. Die neu entstandene 4er-Gruppe hat nun fünf Freiheiten und ist damit nicht mehr unmittelbar vom Tod bedroht.
Weiß bedroht nun mit 10 abermals einen schwarzen Stein. Schwarz flieht mit 11 und die Gruppe hat damit 3 Freiheiten.
Weiß versucht mit 12 die schwarze 2er-Gruppe etwas unter Druck zu setzen, indem er

[3]Die genaue Höhe des Komi variiert je nach Regeln (jap., chin., AGA), Turnier und auch über die Jahre hinweg um ein bis zwei Punkte.
[4]Quelle des Spiels: `http://playgo.to/interactive/`

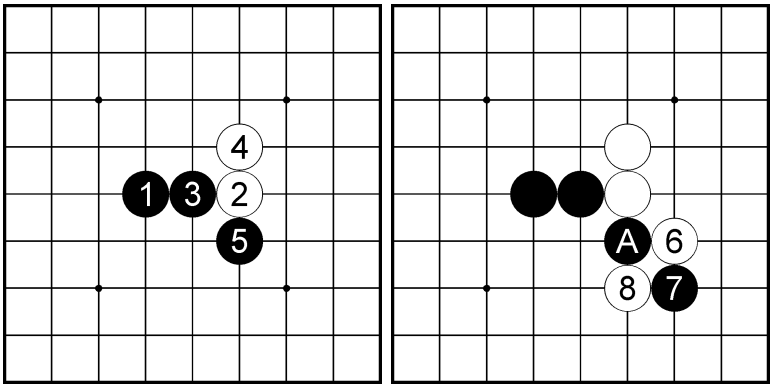


Abbildung B.6.: Zug 1 bis 8

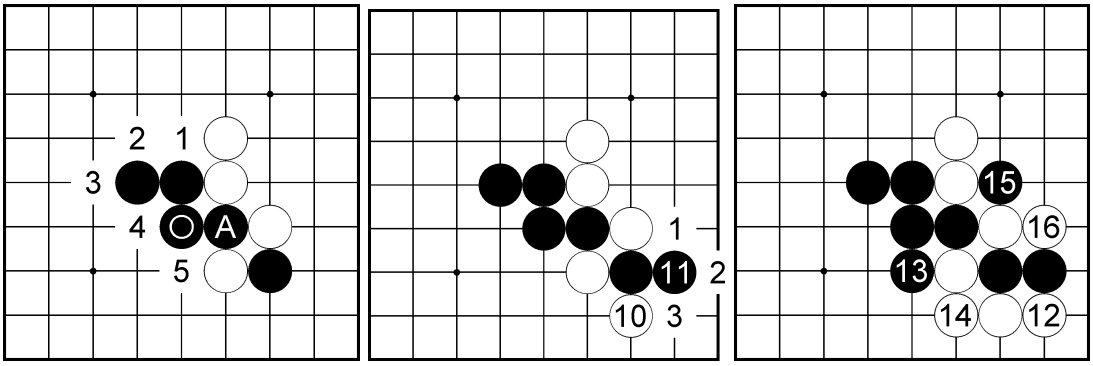


Abbildung B.7.: Zug 9 bis 16

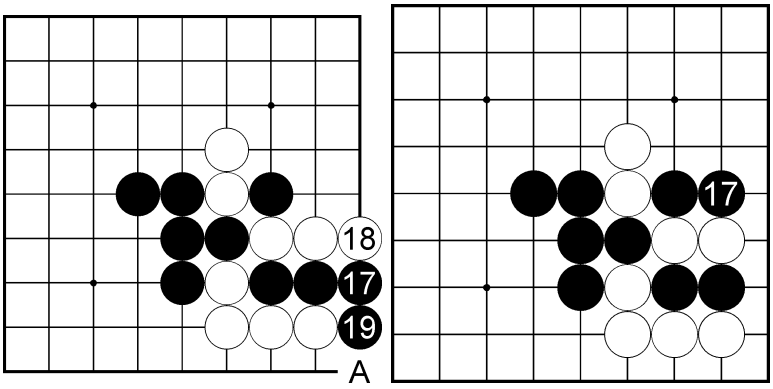


Abbildung B.8.: Variation zu Zug 17 und der tatsächlich gespielte Zug

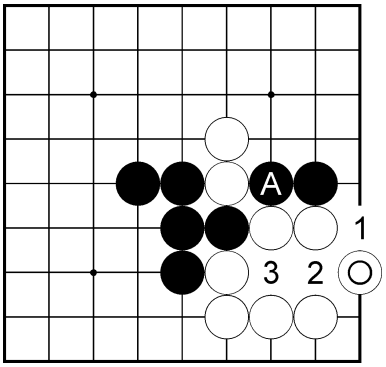


Abbildung B.9.: Zug 18

auf eine ihrer Freiheiten spielt und sie ihnen damit wegnimmt. Schwarz spielt 13 und bedroht nun seinerseits einen weißen Stein. Weiß verbindet diesen daher mit 14 mit den anderen beiden. Wiederum bedroht Schwarz mit 15 einen weißen Stein, Weiß flieht mit 16 und bedroht damit die zwei schwarzen.

Nun könnte Schwarz versuchen, noch weiter an den Rand zu fliehen. Dies wird in Abbildung B.8 in der Variation gezeigt. Die Flucht wäre allerdings wenig aussichtsreich. Weiß hätte ihm hier schnell die verbliebenen Freiheiten genommen und könnte mit A schließlich vier Steine schlagen. Schwarz hätte Weiß zusätzliche Gefangene geschenkt und wertvolle Züge vergeben. Schwarz entscheidet sich daher nicht für die Variation, sondern bedroht mit 17 zwei Steine.

In Abbildung B.9 schlägt Weiß die 2er-Gruppe und die Gruppe, die vorher nur noch eine Freiheit hatte, hat jetzt wieder drei. Man könnte nun überlegen, warum Schwarz in Zug 17 überhaupt diese weiße Gruppe bedroht hatte, wenn sie sich so einfach wieder befreien kann. Schwarz hatte Zug 17 so gespielt, da dieser Zug seinen Einfluss auf die rechte obere Ecke verstärkt und den alleinstehenden, gefährdeten Stein A zusätzlich

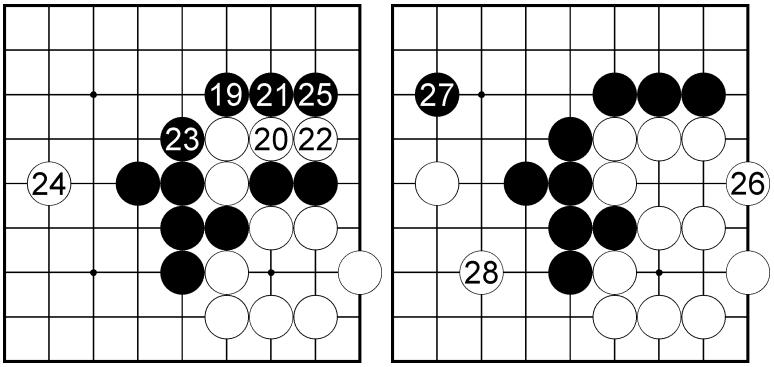


Abbildung B.10.: Zug 19 bis 28

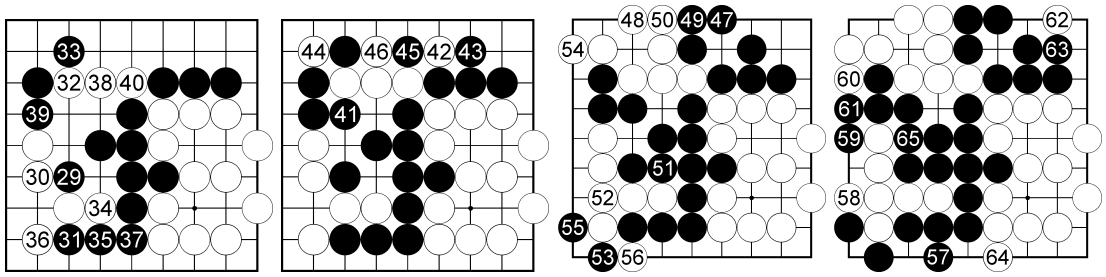


Abbildung B.11.: Zug 29 bis 65

absichert. Man wird zwar gleich sehen, dass A und sein neuer Nachbar dennoch geopfert werden, aber auch dies folgt einem Zweck.

Nach den Zügen 19 bis 23 sieht man in Abbildung B.10, dass lediglich Schwarz Einfluss auf die linke Bretthälfte hat. Weiß spielt daher 24 in die linke Hälfte. Dieser einzelne Stein ist recht wehrlos, Schwarz greift ihn aber zunächst einmal nicht direkt an, sondern beschäftigt erst einmal Weiß mit seinem Zug 25. Weiß schlägt mit 26 die zwei Steine. Schwarz hatte sie geopfert, damit seine gesamte Situation gefestigt ist und kann nun mit 27 entspannt Weiß auf der linken Seite angreifen. Warum er mit etwas Abstand und nicht direkt angreift ist schwerer zu erklären und würde den Rahmen hier sprengen. Weiß fühlt sich nun in dieser Hälfte zu schwach für eine direkte Konfrontation und weicht mit 28 aus.

Verbundene Steine sind stärker als einzelne. Schwarz 29 versucht daher in Abbildung B.11 die drohende Verbindung der zwei weißen Steine zu unterbrechen. Weiß wirkt dem mit 30 entgegen. Es folgen die Züge 31 bis 65.

Nach Zug 66 in Abbildung B.12 liegt eine Ko-Stellung vor. Schwarz darf nicht direkt auf A zurückschlagen. Er spielt daher an anderer Stelle, Weiß reagiert darauf und Schwarz kann mit 69 zurückschlagen. Nun würde Weiß gerne direkt zurückschlagen, muss aber 70

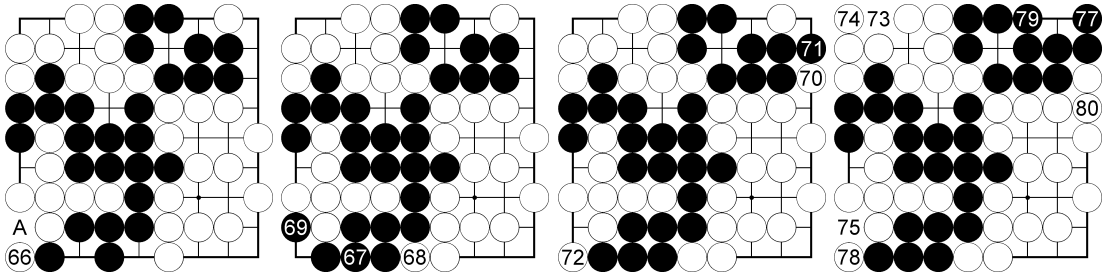


Abbildung B.12.: Zug 66 bis 80

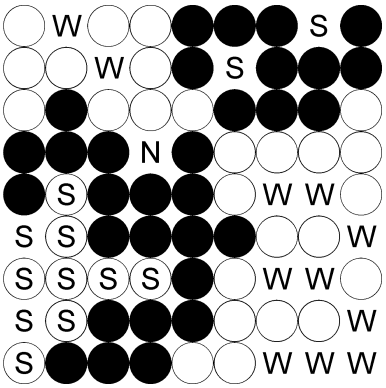


Abbildung B.13.: Spielende und Auszählung

erst an anderer Stelle spielen, Schwarz reagiert darauf und Weiß schlägt mit 72 zurück. Dieser sog. Ko-Kampf geht noch ein paar Züge weiter bis schließlich der letzte Zug 80 gespielt wird. Nach Zug 80 hat Weiß keine Drohungen an anderer Stelle mehr parat, um den Ko-Kampf noch weiter zu führen und passt daher. Schwarz passt auch und das Spiel ist vorbei.

In Abbildung B.13 hat Weiß die 11 mit W markierten Felder umzingelt und im Laufe des Spiels 9 schwarze Steine gefangen. Weiß hat also 11+9+5.5 = 25.5 Punkte.
Schwarz hat 26 Punkte. Ganz exakt gehe ich hier nicht darauf ein. Es sei nur gesagt, dass die weißen Steine in der linken unteren Ecke unrettbar verloren sind und damit auch Schwarz zugeschrieben werden. Schwarz' Bilanz steigt damit um 8 auf 14 Gefangene und um 10 auf 12 Felder.
Das mit N markierte Feld wird von keinem von beiden allein umzingelt und gehört damit niemandem. Schwarz gewinnt also mit einem halben Punkt Vorsprung.
Solche knappen Ergebnisse sind zumindest bei Profis nicht ungewöhnlich. Merkt ein Spieler im Laufe des Spiels, dass er deutlich vorne liegt, steigt er auf eine weniger gierige, dafür aber sicherere Spielweise um und das Ergebnis tendiert damit immer zu einem knappen aber sicheren Sieg.

Allein durch die Kenntnis der Regeln ist es vermutlich weniger als in anderen Spielen möglich ein gutes Spiel zu spielen. Hierzu sind Konzepte wie Augen, Leben und Tod und viele andere mehr nötig. Ein guter Einstiegspunkt für Interessierte ist z.B. `http://senseis.xmp.net/?PagesForBeginners`.
Das Spiel auf 9x9 erweckt den Anschein, dass Go lediglich ein taktischer Schlagabtausch wäre. Auf 19x19 steht den Spielern allerdings wesentlich mehr Platz für strategische Planungen zur Verfügung.

B.7. Ränge und Handicap

Natürlich passiert es häufig, dass Spieler mit unterschiedlicher Spielstärke aufeinander treffen. Spielstärken werden in Graden beurteilt und der Ausgleich von Spielern unterschiedlicher Spielstärke geschieht durch sog. Handicap-Steine. Da Grade auch in der Beurteilung der Stärke von KIs benutzt werden und Handicaps auch zwischen zwei KIs oder einem Menschen und einer KI üblich und auf Grund des großen Spielstärkeunterschieds zwischen Menschen und KIs auch nötig sind, werden sie hier kurz erläutert.
Die Spielstärke wird ähnlich wie in verschiedenen Kampfsportarten bewertet. Man startet ca. mit dem 30. Kyû (30k, 30. Schülergrad), arbeitet sich hoch bis zum 1. Kyû, wird dann 1. Dan (1d, 1. Meistergrad) und kann sich bis ca. zum 8. Dan hocharbeiten. Für asiatische Berufsspieler gibt es zusätzlich noch Profi-Grade.
Um den Unterschied zwischen zwei Spielern auszugleichen, gibt es ein Handicapsytem. Der schwächere Spieler spielt stets Schwarz. Außerdem werden Weiß bei einem Handicapspiel stets die Ausgleichspunkte (Komi) gestrichen. Ist Schwarz einen Grad schwächer, geschieht sonst nichts. Bei zwei Graden, darf er vor Weiß' erstem Zug zwei Steine statt einem spielen, bei drei Graden drei Steine usw..
Über das Handicap-System definieren sich auch die Grade. Braucht man gegen einen 7. Kyû zwei Steine, um im Schnitt ausgeglichen zu spielen, ist man ein 9. Kyû. Handicaps sind zwar auch in anderen Spielen (z.B. Schach) möglich, allerdings ist im Go auch mit Handicap ein relativ natürliches Spiel möglich und der Einsatz von Handicaps ist daher absolut üblich.

Literaturverzeichnis

[1] *The French Federation of Go homologates a victory of a computer against a professional player in 9x9 Go.* – `http://www.lri.fr/~teytaud/crmogo.en.html`

[2] *Sensei's Library-Artikel über MoGo.* – `http://senseis.xmp.net/?MoGo`

[3] Araki, Nobuo ; Yoshida, Kazuhiro ; Tsuruoka, Yoshimasa ; Tsujii, Jun'ichi: Move Prediction in Go with the Maximum Entropy Method. In: *2007 IEEE Symposium on Computational Intelligence and Games*

[4] Boon, M.: A Pattern Matcher for Goliath. In: *Computer Go* (1990), Nr. 13, S. 12–23

[5] Bouzy, Bruno ; Cazenave, Tristan: Computer Go: An AI oriented survey. In: *Artificial Intelligence* 132 (2001), S. 39–103

[6] Bouzy, Bruno ; Chaslot, Guillaume: Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 Go. In: Kendall, G. (Hrsg.) ; Lucas, Simon (Hrsg.): *IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, 176–181

[7] Campbell, Murray ; Hoane, A. J. ; Hsu, Feng hsiung: Deep Blue. In: *Artif. Intell.* 134 (2002), Nr. 1-2, S. 57–83. – ISSN 0004–3702

[8] Chaslot, Guillaume ; Winands, Mark ; Van Den Herik, Jaap H. ; Uiterwijk, Jos ; Bouzy, Bruno: Progressive Strategies for Monte-Carlo Tree Search. In: *Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session*

[9] Churchill, Julian ; Cant, Richard ; Al-dabass, David: *A New Computational Approach to The Game of Go*

[10] Coulom, Rémi: Computing Elo Ratings of Move Patterns in the Game of Go. In: *ICGA Journal* 30 (2007), December, Nr. 4, 198–208. `http://remi.coulom.free.fr/Amsterdam2007/MMGoPatterns.pdf`

[11] Dahl, Fredrik A.: Honte, a Go-Playing Program Using Neural Nets. In: Fürnkranz, Johannes (Hrsg.) ; Kubat, Miroslav (Hrsg.): *Workshop on Machine learning in Game Playing* 16th International Conference on Machine Learning (ICML-99), Bled, Slovenia, Nova Science Publishers, 205–223

[12] Donnelly, Paul ; Corr, Patrick ; Crookes, Danny: *Evolving Go playing strategy in neural networks*. 1994

[13] Enzenberger, Markus: The integration of a priori knowledge into a go playing neural network. 1996. – Forschungsbericht

[14] Gelly, Sylvain ; Silver, David: Combining Online and Offline Knowledge in UCT. In: *International Conference on Machine Learning, ICML 2007*

[15] Gelly, Sylvain ; Wang, Yizao ; Munos, Rémi ; Teytaud, Olivier: Modification of UCT with Patterns in Monte-Carlo Go / INRIA, France. Version: November 2006. `http://hal.inria.fr/docs/00/12/15/16/PDF/RR-6062.pdf` (6062). – Forschungsbericht. – Elektronische Ressource

[16] Heise.de: *Computer-Go: Die Revanche des Profis*. – Heise.de-Artikel über das Spiel zwischen MoGo und einem koreanischen Profi: `http://www.heise.de/newsticker/Computer-Go-Die-Revanche-des-Profis--/meldung/116284`

[17] Kojima, Takuya ; Ueda, Kazuhiro ; Nagano, Saburo: An evolutionary algorithm extended by ecological analogy and its application to the game of go. In: *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97*, Morgan Kaufmann, 1997, S. 684–689

[18] Mayer, Helmut A. ; Maier, Peter: Evolution of Neural Go Players. In: *ÖGAI Journal 2005* (2005)

[19] Müller, Martin: Computer Go. In: *Artif. Intell.* 134 (2002), Nr. 1-2, S. 145–179. `http://dx.doi.org/http://dx.doi.org/10.1016/S0004-3702(01)00121-7`. – DOI http://dx.doi.org/10.1016/S0004–3702(01)00121–7. – ISSN 0004–3702

[20] Samuel, A. L.: Some studies in machine learning using the game of checkers. In: *IBM Journal of Research and Development* 3 (1959), Nr. 3, 210–229. `http://www.research.ibm.com/journal/rd/033/ibmrd0303B.pdf`

[21] Simon, Herbert A. ; Schaeffer, Jonathan: The Game of Chess. In: *Handbook of Game Theory, volume I of Handbooks in Economics, v*, Elsevier Science B.V, 1992, S. 1–17

[22] Stern, David ; Herbrich, Ralf ; Graepel, Thore: Bayesian Pattern Ranking for Move Prediction in the Game of Go. In: *International Conference on Machine Learning (ICML-2006)*

[23] Tesauro, Gerald: Programming backgammon using self-teaching neural nets. In: *Artificial Intelligence* 134 (2002), S. 181–199

[24] Van Den Herik, Jaap H. ; Uiterwijk, Jos W. H. M.: Solving Go on small boards. In: *International Computer Games Association Journal* 26 (2003), S. 10–7

[25] Van Der Werf, Erik ; Uiterwijk, Jos ; Postma, Eric ; Van Den Herik, Jaap: *Local move prediction in Go*

[26] Wikipedia: *Computer Go — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=Computer_Go&oldid=294763898`. Version: 2009. – [Online; accessed 6-June-2009]

[27] Wolf, Thomas: The Program GoTools and its Computer-generated Tsume Go Database. In: *In Proceedings of the First Game Programming Workshop*, 1994, S. 84–96

[28] Zobrist, Albert L.: A new hashing method with application for game playing / Computer Science Department, University of Wisconsin. 1970. – Forschungsbericht