Entwicklung und Vergleich von maschinellen Spielern für Gomoku, Gobang und Pente

Abschlussarbeit im Studiengang:

Bachelor 1-Fach Informatik

Sommersemester 2012

Vorgelegt von:

Arne Kowalewski

Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion

Christian-Albrechts-Universität zu Kiel

Betreuer: Prof. Dr. Michael Hanus

Zweiter Gutachter: Dr. Friedemann Simon

Abgabedatum: 28. 09. 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Kurzfassung

Die Arbeit befasst sich mit der Entwicklung von maschinellen Spielern[1] für das Brettspiel Gomoku und seinen Variationen Gobang und Pente. Die Brettspiele und ihre Regeln werden zu Beginn erklärt. Anschließend wird das Problem selbst analysiert, wobei mit Problem die zu bearbeitende Thematik gemeint ist. Aus dieser Analyse gehen zwei Verfahren hervor, die sich zur Lösung der Aufgabenstellung eignen. Bei diesen handelt es sich zum einen um das *Influence Mapping* und zum anderen um die Suche in *Spielbäumen*. Beide Verfahren werden explizit erklärt und Optimierungsansätze werden genannt. Da beide Verfahren Heuristiken zum Ermitteln der bestmöglichen Züge verwenden, wird das Konzept der Heuristik erläutert. Außerdem wird darauf eingegangen, was eine gute Heuristik ausmacht.
Im Anschluss wird die technische Umsetzung der beschriebenen Verfahren detailliert beschrieben. Dabei wird sowohl auf die Funktionsweise des Verfahrens, als auch auf die Heuristik selbst, sowie Vor- und Nachteile des Ganzen eingegangen. Die Umsetzung umfasst gleichzeitig den innovativen Anteil der Arbeit. Die verwendeten Heuristiken wurden eigenständig entwickelt und an die verwendeten Verfahren angepasst. Auch Optimierungen an den Verfahren gehören zum innovativen Anteil. Die Kombination dieser Bestandteile und die daraus entstehenden Spieler sind also eigenständig entwickelt und finden sich in dieser Form nirgendwo sonst.
An die Beschreibung der Umsetzung schließen sich praktische Tests der Implementierung an. Auch Vergleiche mit anderen maschinellen Spielern, welche das selbe Problem lösen, werden beschrieben und ihre Ergebnisse werden erläutert. Abschließend wird noch einmal ein Fazit zur Arbeit mit den entwickelten Spielern und den Verfahren bezüglich der Aufgabenstellung gezogen.

[1] Im Folgenden auch als Agenten bezeichnet

2 Einleitung

Diese Arbeit wurde als Abschlussarbeit für den Studiengang *Bachelor der Informatik* verfasst. Die Arbeit befasst sich mit der Entwicklung von maschinellen Spielern, welche auch als Agenten bezeichnet werden. Für die Arbeit werden zwei vergleichsweise einfache Verfahren aus dem Bereich der Agenten-Entwicklung implementiert und verglichen werden. Als Umgebung wurden drei Brettspiele gewählt. Im restlichen Teil des Kapitels werden die Brettspiele erklärt und es wird darauf eingegangen, was einen intelligent spielenden Agenten ausmacht. Abschließend wird das Ziel der Arbeit erläutert.

2.1 Die Brettspiele

Brettspiele sind Spiele, deren kennzeichnendes Element ein Spielbrett ist, auf dem Spieler mit Figuren oder Steinen agieren. Die Prinzipien solcher Brettspiele werden im Rahmen dieser Arbeit auf die moderne Technik übertragen. Die Brettspiele sollen also durch eine Software umgesetzt werden. Die Software ermöglicht das Spielen zweier Spieler, wobei die Maus als Eingabegerät verwendet wird. Da es sich um Brettspiele handelt, wird selbstverständlich auch ein Spielbrett benötigt. Bei den Brettspielen, die mit der Software umgesetzt wurden, handelt es sich um zug-basierte Spiele. Alle im Folgenden vorgestellten Spiele sind Zwei-Spieler-Spiele. Die Spieler sind dabei immer abwechselnd am Zug.
Der erste Grund dafür, dass alle drei statt nur eines der im Folgenden erklärten Spiele implementiert wurden, ist die Frage nach der Flexibilität der Implementierung. Da sich die drei Spiele nur durch geringe Regeländerungen unterscheiden, soll gezeigt werden, dass die Agenten diese Änderungen einfach handhaben können.
Der zweite Grund für die Arbeit mit drei Spielen ist, dass nur sehr wenige Vergleichsmöglichkeiten für jedes einzelne dieser Spiele zur Verfügung stehen.

2.1.1 Gomoku

Das Wort Gomoku (auch Go-Moku) kommt aus dem Japanischen und stammt dort von dem Wort *gomokuarabe* ab. *Go* bedeutet fünf, *moku* ist ein Aufzählungsbegriff für Stücke oder Spielsteine und *arabe* bedeutet aneinander reihen. Gomoku ist in englischsprachigen Ländern auch als “Five-In-A-Row“ bekannt[2]. Früher wurde es auf 19x19 Go-Brettern gespielt. Heutzutage werden auch kleinere Bretter mit dem selben Muster verwendet. Ein Go-Brett

[2]http://en.wikipedia.org/wiki/Gomoku(15.8.2012)

besteht aus vertikalen und horizontalen Linien, welche sich kreuzen. Die Kreuze im hierdurch entstehenden Gitter stellen die gültigen Spielpositionen dar, nicht wie beim Schach die Felder selbst. Das Spielfeld ist, wie bei den meisten Brettspielen üblich, zweidimensional.
Das Spiel gehört zur Familie der "Verbinde k"-Spiele, zu denen auch *Tic Tac Toe* und *Vier Gewinnt* zählen [**?**]. Das Ziel ist es, zuerst fünf eigene Spielsteine vertikal, horizontal oder diagonal aneinanderzureihen. Zu den Standard Gomoku-Regeln gehört auch, dass eine Reihe, deren Länge mehr als fünf Spielsteine beträgt, nicht als Sieg zählt (*Overline*). Für die Implementierung der Software wird die Overline-Regel aber nicht verwendet, da zu Testzwecken die Regeln flexibler gehalten werden müssen. Weiterhin ist es bei diesem Spiel nicht möglich Steine vom Spielfeld zu entfernen, welche einmal gesetzt wurden. Das Spiel selbst wird als unentschieden gewertet, falls keine Position zum Setzen eines Steines mehr frei ist und keiner der Spieler eine Fünfer-Reihe besitzt. Aufgrund seiner einfachen Regeln wird dieses Spiel häufig von Kindern gespielt doch gilt hier: „Leicht zu lernen, doch schwer zu meistern." [**?**]

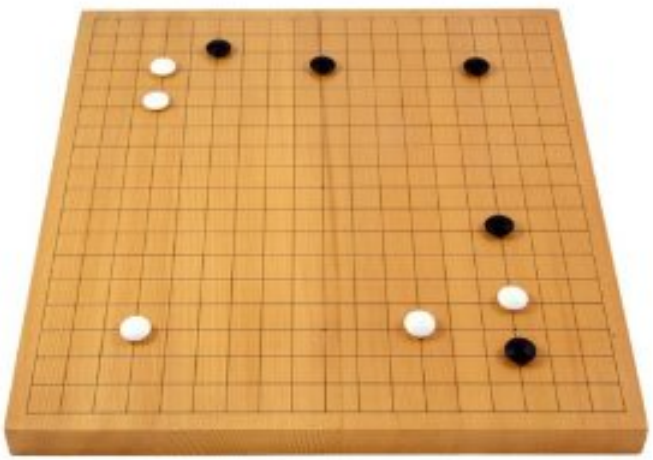

Abbildung 1: Original Go-Brett mit Spielsteinen

2.1.2 Gobang

Gobang ist eine Spielvariante von Gomoku, welche es erlaubt bei bestimmten Spielstellungen Steine des Gegners zu schlagen und diese dann vom Spielfeld zu entfernen. Diese Spielstellungen sind alle nach dem selben Schema aufgebaut:

- Zwei gegnerische Spielsteine bilden eine Zweier-Reihe
- Ein oder mehrere eigene Spielsteine begrenzen diese Reihe auf einer Seite
- Das andere Ende der Reihe ist frei

Wird nun ein eigener Spielstein an das freie Ende der Zweier-Reihe gesetzt, so werden die zwei gegnerischen Spielsteine vom Spielfeld entfernt. Die Zweier-Reihen können sowohl horizontal, vertikal wie auch diagonal geschlagen werden. Die folgende Abbildung dient zur Verdeutlichung der Schlagmechanik.

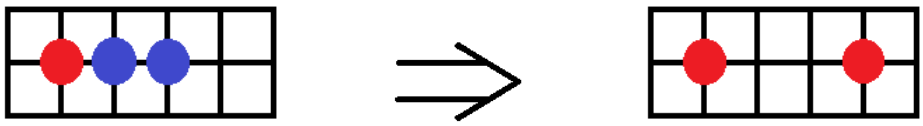

Abbildung 2: Schlagregel Gobang

Die Schlagmöglichkeit gilt allerdings nur für Zweier-Reihen. Es ist also möglich die eigenen Spielsteine von dem Entfernen durch den Gegenspieler zu schützen, indem man die Zweier-Reihe um einen Stein erweitert.
Ein solches Entfernen der Steine ist auch nur durch das aktive Setzen des Steines an das freie Ende der Reihe möglich. Nach dem Entfernen der Steine kann der Gegner also wieder zwei eigene Steine in die entstandene Lücke setzen und diese werden nicht automatisch wieder geschlagen. Mit einem Stein mehrere Zweier-Reihen auf einmal zu schlagen ist dabei legal.

2.1.3 Pente

Pente ist eine weitere Spielvariante, welche von Gomoku abstammt. Mitte der 70er Jahre vereinfachte Gary Gabrel die Regeln von Ninuki-renju, ohne dabei dessen Komplexität zu mindern. Er verwendete den *Fünf in eine*

Reihe-Ansatz von Gomoku und eine der *Schlagregeln* des alten asiatischen Spiels Go und nannte sein neues Brettspiel „Pente“. Pente selbst ist das altgriechische Wort für fünf. In den 70er- und 80er-Jahren war das Spiel sehr populär[3].
Es wird für gewöhnlich auf einem 19x19 Go-Brett gespielt. Es gibt verschiedene Regeln für Pente. Zum Beispiel Turnier-Pente, Keryo-Pente oder No-Capture-Regeln. Für die Implementierung der Software werden die Freestyle-Regeln verwendet. Diese Regeln sind die selben wie die von Gobang, wobei bei einer Pente-Runde die geschlagenen Steine gezählt werden, wodurch eine zweite Möglichkeit besteht, zu gewinnen. Ein Spieler gewinnt in diesem Spiel auch, falls er zuerst zehn gegnerische Steine geschlagen hat. Pente kann auch von mehr als zwei Spielern gleichzeitig gespielt werden. Die Variante mit zwei Spielern ist jedoch die am meisten verbreitete und wird auch von dieser Software unterstützt.
Beim Spielen von Pente ist es besonders wichtig, ein Auge auf die Zweier-Reihen zu haben, da fünf geschlagene Zweier-Reihen auch zum Sieg führen. Ergibt sich eine Zweier-Reihe, sollte diese umgehend verlängert oder (falls sie dem Gegenspieler gehört) angegriffen werden. Dadurch schränkt man die Zugmöglichkeiten des Gegenspielers ein, falls dieser nicht frühzeitig in Bedrängnis geraten will, durch diese Regel zu verlieren. Hinzu kommt noch, dass es nützlich ist, viele Spielsteine auf dem Feld zu haben, da sich aus mehr Steinen auch mehr Möglichkeiten für eine Fünfer-Reihe ergeben [4]. Obwohl Pente die gleichen Spielmöglichkeiten wie Gobang bietet, ist es also dennoch komplexer.

2.2 Wann spielt jemand intelligent?

Da die Intelligenz als solche noch keine treffende Definition hat, beschränken wir uns in diesem Unterkapitel darauf zu beschreiben, wann ein Agent so spielt, dass ein menschlicher Betrachter ihn als intelligent bezeichnen würde. Ein Agent wird im Allgemeinen als intelligent angesehen, wenn er konsequent Entscheidungen trifft, die man als passend, bezüglich eines gegebenen Kontextes, einstuft. Man könnte also sagen, dass das Treffen von Entscheidungen der Kern eines „intelligenten“ Agenten ist [**?**].
Wir bezeichnen *passend* spielen, bezüglich einen gegebenen Kontextes, im Folgenden als *nachvollziehbar* spielen. Im Fall der beschriebenen Brettspiele

[3] http://boardgamegeek.com/thread/72573/a-simple-twist-on-an-ancient-game (16.8.2012)

[4] http://www.yourturnmyturn.com/rules/pente.php (17.8.2012)

würde ein Betrachter es sicherlich als nachvollziehbar empfinden, wenn der Agent versucht, eine Fünfer-Reihe zu bauen oder eine Vierer-Reihe des Gegners zu blockieren, um eine Niederlage zu vermeiden. Für einen Computer ist es aber schwieriger als für menschliche Spieler, gute Spielstellungen zu erkennen oder sich eine Folge von Zügen zu überlegen.
Sowohl Mensch als auch Maschine benötigen einige Zeit um sich eine gewinnbringende Folge von Zügen zu überlegen. Menschen nutzen Methoden der optimistischen Suche und Bestätigung, Programme benutzen koventinelle Baum-Such-Techniken [**?**]. Ein Großteil der Implementierung beschäftigt sich also damit, wie der Agent gute Züge von schlechten unterscheidet.

2.3 Ziel der Arbeit

Nachdem zuvor die Spiele, mit denen gearbeitet wird, vorgestellt wurden und abstrakt erläutert wurde, wann sich ein Agent zum Spielen dieser als „intelligent“ bezeichnen lässt, werden nun die Einzelteile zusammengefügt. Durch die gegebenen Spiele verfolgt diese Arbeit einen problemspezifischen Ansatz. Das Ziel dieser Arbeit ist die Entwicklung eines *nachvollziehbar spielenden* Agenten für die Spiele Gomoku, Gobang und Pente. Die Entwicklung schließt die Implementierung der Spiele und eine dazugehörige Benutzeroberfläche mit ein. Schwerpunkt der Arbeit ist jedoch die Entwichlung von Verfahren zur Zugauswahl mittels dazu passenden Heuristiken.

3 Problemanalyse und Verfahren zur Lösung

Die Forschungen zur künstlichen Intelligenz stellen einige Verfahren, zum Erstellen von Agenten zur Verfügung [**?**]. Um geeignete Verfahren zu wählen, muss zuerst das Problem selbst eingeordnet werden, denn nicht jedes Verfahren eignet sich für jedes Problem. Bei den bisher genannten Spielen handelt es sich um Spiele *für zwei Personen*. Heuristische Algorithmen für diese Spiele sind komplizierter als jene für beispielsweise einfache Puzzles, da ein im Wesentlichen unberechenbarer Gegenspieler existiert [**?**].
Des Weiteren werden die Spiele *zug-basiert* und nicht in Echtzeit gespielt. Es reicht also aus, den Feldzustand nach jedem Zug zu aktualisieren, da zwischenzeitlich keine Änderungen an diesem möglich sind.
Außerdem sind in diesen Spielen *keine Zufallselemente* enthalten. Mit anderen Worten, Ereignisse wie Würfeln oder das Aufdecken von Karten, welche direkten Einfluss auf das Spiel nehmen (beispielswiese Ereigniskarten bei Monopoly), sind nicht Bestandteile dieser Spiele. Somit ist es einfacher für einen

Agenten den Verlauf des Spiels vorherzusehen.
Als letzter Aspekt wäre noch zu nennen, dass es sich bei allen drei Spielen um Spiele mit *perfektem Wissen* handelt. Perfektes Wissen bedeutet, dass beide Spieler zu jedem Zeitpunkt die selben Informationen über den Spielzustand haben. Ein Gegenbeispiel dazu wäre Poker, bei dem jeder Spieler nur sein eigenes Blatt kennt. Aufgrund des Umfangs dieser Arbeit werden nur zwei Verfahren zur Bearbeitung des beschriebenen Problems ausgewählt.
Zuerst wird das Verfahren des *Influence Mapping* erläutert. Dieses Verfahren bietet noch weitaus umfangreichere Anwendungsmöglichkeiten, wird aber im Rahmen der Arbeit nur zum Planen der Folgezüge verwendet. Danach werden die *Suche im Spielbaum* und die dazu verwendeten Algorithmen erklärt. Daher wird eine Form des Minimax-Algorithmus mit Alpha-Beta-Suche verwendet. Dieses Verfahren ist das wohl bekannteste [**?**], wenn es um die Entwicklung von Brettspiel-Agenten geht. Eine Breitensuche bietet sich für Spielbäume mit großem Umfang nicht an, da sie zu viel Zeit zum Finden einer Lösung benötigen würde. Das *Influence Mapping* wurde gewählt, da es sehr flexibel ist und sich an viele Arten von Problemen anpassen lässt. Die *Suche in Spielbäumen* wurde gewählt, da es das meistverwendete Verfahren bezüglich der Entwicklung von Agenten für Brettspiele ist und mit den genannten Algorithmen Performance-Vorteile gegenüber anderen Suchbaum-Verfahren wie dem A*-Algorithmus oder dem British Museum hat [**?**].
Neben den gewählten Verfahren wurden auch Überlegungen bezüglich einer Entwicklung mittels *Expertensystemen* getätigt. Bei diesem hätte sich ein fallbasiertes System angeboten, welches über eine Falldatenbank verfügt, in der konkrete Problemstellungen inklusive einer Lösung gespeichert sind. Dieser Ansatz wurde allerdings wieder verworfen, da bei den vorgestellten Brettspielen eine unvorstellbar große Anzahl an Problemstellungen generiert werden kann, die Datenbank extrem umfangreich wäre und das Durchsuchen dieser zu lange dauern würde.

4 Influence Mapping

Das Influence Mapping ist ein Verfahren, welches seinen Ursprung in der Robotik hat. Es wurde erstmals von O. Khatib [**?**] zur Echtzeit-Kollisionsvermeidung von mobilen Robotern eingeführt. Dieses Verfahren wird allerdings auch zur Entwicklung von Agenten in Spielen eingesetzt. Der Grund für den Einsatz dieses Verfahrens ist die Möglichkeit, schnell auf Änderungen in der Umgebung des Agenten zu reagieren, wobei in diesem Fall die Umgebung des Agenten das Spielbrett ist. Influence Mapping kann sowohl in Strategie-

Spielen, First-Person-Shootern oder Brettspielen angewendet werden [**?**].
Wir **definieren** Influence Maps folgendermaßen:
Eine *Influence Map* ist eine Abbildung der Umgebung. Die Umgebung wird dafür in Form einer zweidimensionalen Matrix dargestellt. Jede Zelle der Matrix enthält dabei Informationen über die jeweiligen Gegebenheiten der Umgebung[5].
Häufig verfügt jeder Agent über eine Influence Map für sich selbst und eine für jeden anderen Spieler [**?**]. Die Vorteile dieses Verfahrens sind zum einen, dass dem Agenten selbst nicht viele Nachrichten, bezüglich der Umgebung, von außerhalb geschickt werden müssen und zum anderen sind die verwendeten Funktionen eine Reaktion auf die Änderungen der Umgebung, statt dem Folgen einer festen Menge von Regeln [**?**].
Die Zellen der Influence Map können die verschiedensten Informationen enthalten. In Strategiespielen werden beispielsweise Daten über Ressourcen, Kampfkraft, Wetter oder Gefahrenpotential gespeichert. Eine Zelle der Inluence Map entspricht im Programm einer Position, auf welche ein Stein gesetzt werden kann. Beim Influence Mapping ist es auch möglich die Werte von Zellen an benachbarte Zellen zu propagieren, um die Bewegungen feindlicher Einheiten besser vorhersehen zu können. Diese Propagierung ist linear, exponentiell und auf noch weitere Arten möglich[6]. Dieser Aspekt wurde aber in der Arbeit nicht verwendet, da Steine ohne Einschränkungen überall hin gesetzt werden können. Dadurch ist es schwer das Potential eines einzelnen Steines für die umliegenden Positionen einzuschätzen.
Die Entwicklung für die genannten Spiele lässt sich daher in zwei Teile gliedern. Zuerst wird ein Modell für die Visualisierung des Spielfeldes, analog zur menschlichen Sicht des Spielers, generiert. Danach werden Heuristiken gesammelt, die den kognitiven Prozessen von Spielern ähneln [**?**]. Die Influence Map muss dann in sinnvollen zeitlichen Abständen aktualisiert werden, um den Spieler auf den neuesten Stand zu bringen. Damit wird der Einfluss von Objekten auf in der Nähe befindliche Bereiche der Karte gezeigt.
Das folgende Beispiel einer Influence Map von Maurice Bergsma und Peter Meij [**?**] soll zur Illustration dienen:

[5] http://de.slideshare.net/mobius.cn/influence-map (12.9.2012)
[6] http://de.slideshare.net/mobius.cn/influence-map (12.8.2012)

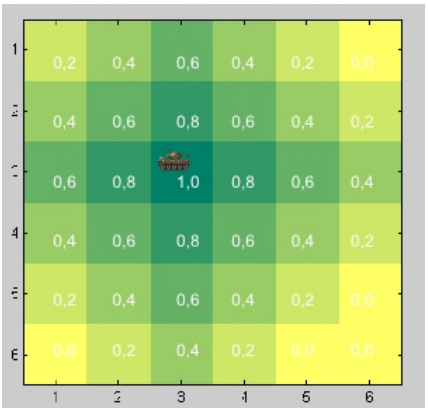

Abbildung 3: Influence Map - Angriffskraft eines Panzers

Bergsma und Meij [**?**] fassen Folgendes für Influence Maps zusammen:

- Influence Maps sind ein intuitiver Ansatz zur Beschreibung von Spielfeldern.
- Sie sind verhältnismäßig einfach zu implementieren.
- Sie benötigen viel Feinabstimmung um gute Resultate zu erzielen.
- Sie können effektiv zur Entscheidungsfindung genutzt werden.

5 Spielbäume

Das Konzept der Spielbäume ist das wohl am häufigsten verwendete in der Entwicklung von Brettspielen [**?**]. Es beruht auf der Herangehensweise eines menschlichen Spielers, Züge im Voraus zu planen. Für Spielbäume wird folgende **Definition** verwendet:
Die Wurzel eines Spielbaums ist die aktuelle Stellung der Steine auf dem Spielbrett. Die Knoten sind Spielzustände des Spielbretts und die Kanten sind den Regeln des jeweiligen Spiels entsprechend legale Züge von einem Spielzustand zu einem anderen.
Es gibt zwei Varianten der Baumgenerierung. Zum einen kann ein Agent vor dem allerersten Zug einen großen Baum konstruieren, welcher effektiv zum Spielen mehrerer Züge eingesetzt werden kann und jedes mal, nachdem der

Baum nicht mehr zu gebrauchen ist, einen neuen eben so großen berechnen. Zum anderen kann zu Beginn jedes eigenen Zuges ein kleinerer Baum mit dem aktuellen Spielbrettzustand als Wurzel berechnet werden. Letztere Variante wird meistens verwendet, da der Baum aktueller ist und kein langes Warten nach einer bestimmten Zuganzahl von Nöten ist.
In einem solchen Baum muss jeder Spieler davon ausgehen, dass sein Gegenspieler sein bestmögliches versucht, um das Spiel zu gewinnen, oder Aktionen des Spielers zu verhindern. In einem Spielbaum für zwei Spieler ziehen beide Spieler abwechselnd pro Ebene. Solche Spielbäume eignen sich nur für deterministische Spiele, da sonst kein zu erreichendes Ziel angegeben werden kann und eine dazugehörige Heuristik nicht weiß, wonach sie suchen soll. Zur Visualisierung dient das folgede Beispiel eines Spielbaumes für das Brettspiel Tic Tac Toe.

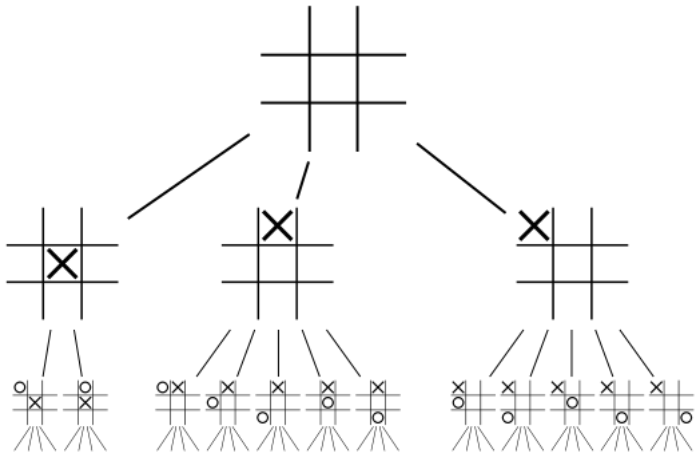

Abbildung 4: Spielbaum: Tic Tac Toe

Die größte Schwierigkeit besteht bei einem Spielbaum darin, den Suchraum sinnvoll einzuschränken, um den Spielbaum möglichst weit in die Tiefe wachsen zu lassen. Je mehr Ebenen der Spielbaum hat, um so besser spielt der maschinelle Spieler im Normalfall auch [**?**], denn eine Ebene im Spielbaum entspricht einem vorausgeplanten Zug. Obwohl das Spielprinzip dem von Tic Tac Toe und Vier-Gewinnt sehr ähnelt, ist der Suchraum von Gomoku, Gobang und Pente um ein Vielfaches größer. Während Tic Tac Toe auf einem 3x3 Feld gespielt wird, hat der Spielbaum einen unoptimierten Verzweigungsgrad von 9!. Wird beim Schach der Spielbaum aus der Startposition des Spiels aufgebaut, enthält er zum Beispiel weit über 10^{40} Knoten [**?**]. Die Anzahl der Knoten in einem Spielbaum ist ungefähr W^D, wobei W

für den durchschnittlichen Verzweigungsgrad und D für die durchschnittliche Anzahl der Züge bis zum Spielende steht. Für Gomoku, Gobang oder Pente auf einem 19x19-Go-Brett wäre W ungefähr 180, und falls nur ein Viertel des Spielfeldes bis zum Sieg einer Spielers gefüllt wäre, hätte D eine Größe von 90. Nach Qian Liang [**?**] müsste man mit einer Anzahl von 180^{90} Knoten rechnen. Spielbäume mit dieser Größe lassen sich von keinem Programm der Welt erschöpfend durchsuchen. Es müssen also Möglichkeiten gefunden werden, einen solchen Spielbaum derart einzuschränken, dass eine *Suche in angemessener Zeit* möglich ist.

5.1 Sinnvolle Beschränkungen

Aufgrund der extrem großen Knotenanzahl eines Spielbaumes ist es zwingend notwendig, den Baum einzuschränken, um eine *akzeptable* Suchtiefe zu erreichen. Für Bäume im zuvor genannten Umfang wird eine Tiefe zwischen drei und sieben angestrebt. Dabei besteht unter anderem die Möglichkeit, den Verzweigungsgrad des Baumes einzuschränken. Des Weiteren ist es ratsam, komplette Zweige des Baumes *abzuschneiden*, falls diese keine vielversprechenden Zugmöglichkeiten enthalten. Ein Algorithmus dafür wird im Kapitel 5.2 erklärt. Schlussendlich ist es auch noch sinnvoll die Vertiefung des Baumes abzubrechen, falls sich das Spielfeld bereits in einem Zustand befindet, in dem einer der beiden Spieler gewonnen hat. In einem solchen Zustand ist es nicht mehr nötig weitere Folgezüge zu betrachten, da das Spiel bereits beendet ist. Wendet man diese Beschränkungen an, so können in einem Baum mit einer sehr hohen Wahrscheinlichkeit weitere Ebenen durchsucht werden.

5.2 Suche im Spielbaum

Ist der Spielbaum erst einmal erstellt, muss dieser nach einem möglichst guten Zug durchsucht werden. Für Spiele mit den genannten Eigenschaften gibt es Algorithmen, die das Erstellen des Baumes und das Bewerten seiner Knoten während einer Zugberechnung übernehmen. Diese Algorithmen lassen sich allerdings nur auf *erschöpfend durchsuchbare* Spielbäume anwenden, weshalb die in Kapitel 5.1 beschriebenen Beschränkungen zwingend notwendig sind [**?**]. Ein Spielbaum wird als erschöpfend durchsuchbar bezeichnet, wenn für eine Suche in endlicher Zeit ein Ergebnis geliefert wird. Im Folgenden werden die dazu verwendeten Algorithmen erklärt.

5.2.1 Der Minimax-Algorithmus und Negamax-Optimierung

Der bekannteste und am häufigsten verwendete Algorithmus zum Erstellen eines Spielbaumes ist der Minimax-Algorithmus[**?**]. Der Algorithmus geht von der Annahme aus, dass beide Spieler über das gleiche Wissen bezüglich des Spielzustandes verfügen und ihr Bestes geben, um zu gewinnen. Die beiden Spieler in einem solchen Spiel werden MIN und MAX genannt. [...] Die Bedeutung der Namen ist sofort ersichtlich: MAX repräsentiert den Spieler, der gewinnen, bzw. seinen Vorteil maximieren möchte. MIN ist der Gegner, der versucht, das Spielergebnis von MAX zu minimieren [**?**]. MIN verwendet dabei die gleichen Informationen wie MAX und versucht immer zu einem Zustand zu gelangen, der für MAX ungünstiger ist.
Bei der Implementierung des Algorithmus wird jede Ebene des Suchraums danach benannt, wer zu diesem Zeitpunkt des Spiels am Zug ist. Ist der Suchraum erschöpfend durchsuchbar, werden die Blätter des Baumes mit 1 bewertet, falls MAX gewinnt und mit 0 falls MIN gewinnt. Gilt dies nicht, so kann der Baum nur bis zu einer vordefinierten Anzahl von Ebenen durchsucht werden. Diese Strategie nennt sich *Vorausschauen über n Züge*, wobei n die Anzahl der zu untersuchenden Ebenen bezeichnet [**?**]. Die Anzahl der berechneten Ebenen hängt von Speicher- und Zeiteinschränkungen der Computers ab. Da die Blätter des Baumes nicht immer Endzustände des Spiels sind, können ihnen nur selten Werte zugewiesen werden, die Gewinn oder Verlust wiederspiegeln. Diese Werte werden von einer heuristischen Funktion zugewiesen (siehe Kapitel 6). Die Funktionsweise des Minimax-Algorithmus wird mittels Pseudocode verdeutlicht:

Listing 1: Pseudocode: Minimax-Algorithmus

```
funktion Max (restTiefe) returns Integer
var größtmöglich, zugWert : Integer

  größtmöglich := - unendlich
  für alle möglichen folgeZüge

    if restTiefe == 0
    then zugWert := heuristische Bewertung
    else zugWert := Min ( restTiefe - 1 )
```

```
    if zugWert > bestmöglich then
      größtmöglich := zugWert

  return größtmöglich


funktion Min (restTiefe) returns Integer
var kleinstmöglich, zugWert : Integer

  kleinstmöglich := + unendlich
  für alle möglichen folgeZüge

    if restTiefe == 0
    then zugWert := heuristische Bewertung
    else zugWert := Max ( restTiefe - 1 )

    if zugWert < kleinstmöglich then
      kleinstmöglich := zugWert

  return kleinstmöglich
```

Die Werte werden gemäß der folgenden Regeln über aufeinanderfolgende Vaterknoten durch den Baum bis zur Wurzel übertragen:

- Ist der übergeordnete Knoten ein MAX-Knoten, erhält er den maximalen Wert, den eines seiner Kinder besitzt.
- Ist der übergeordnete Knoten ein MIN-Knoten, erhält er den minimalen Wert, den eines seiner Kinder besitzt.

Der zugewiesene Wert eines jeden Zustandes zeigt also den besten Zustand an, auf den dieser Spieler hoffen kann (vorausgesetzt, der Gegner spielt so, wie es der Minimax-Algorithmus vorhersagt) [**?**]. Die hergeleiteten Werte dienen zur Wahl zwischen den möglichen Zügen. Die Anwendung des Algorithmus wird noch einmal grafisch in der folgenden Abbildung verdeutlicht:

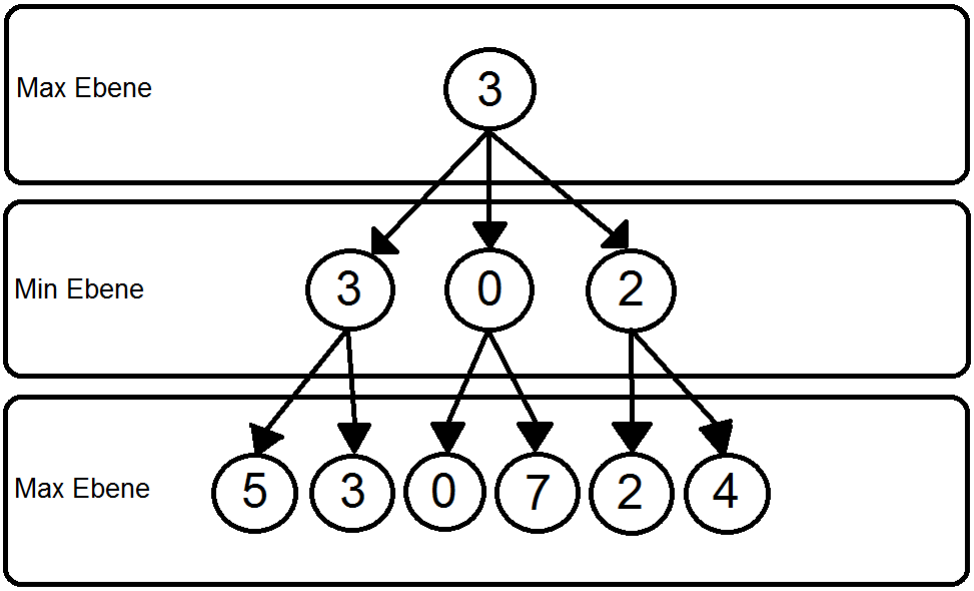

Abbildung 5: Minimax-Algorithmus an einem hypothetischen Baumes

Die **Negamax**-Optimierung ist keine Optimierung im Sinne einer Performance-Verbesserung. Jedoch bewirkt der Negamax eine Vereinfachung des Programmcodes. Während der Minimax-Algorithmus häufig aus einer Funktion für die MIN-Ebenen und einer für die MAX-Ebenen besteht, vereinigt der Negamax-Algorithmus diese geschickt in einer Funktion der selben Länge. Die Idee basiert darauf, die abwechselnde Minimierung und Maximierung durch ein ständiges Maximieren negierter Werte zu ersetzen:
$max\{min\{x1,\dots\}, min\{y1,\dots\}\} = max\{-max\{-x,\dots\}, -max\{-y1,\dots\}\}$
Auf diese Weise müssen bei jedem weiteren Aufruf nur die Vorzeichen vertauscht werden, statt zwischen zwei Funktionen zu wechseln [**?**]. Die Änderungen zum Minimax-Algorithmus werden im folgenden Pseudocode verdeutlicht:

Listing 2: Pseudocode: Negamax-Algorithmus

```
funktion NegaMax (restTiefe, spieler) returns Integer
var größtmöglich, zugWert : Integer
```

```
if restTiefe == 0
  then return spieler * heuristische Bewertung
      /* spieler ist 1 oder -1 */

für alle möglichen folgeZüge
  zugWert := -negaMax(restTiefe -1, -1*spieler)

  if zugWert > bestmöglich then
    größtmöglich := zugWert

return größtmöglich
```

5.2.2 Alpha-Beta-Suche

Die Alpha-Beta-Suche ist eine Erweiterung für den Minimax-Algorithmus. Das Ziel ist es, beim frühzeitigen Erkennen von Zweigen des Baumes, welche das Ergebnis der Suche nicht beeinflussen, diese nicht weiter zu verfolgen. Ist der Verzweigungsgrad eines Baumes besonders groß, so gestaltet sich das einfache Anwenden des Minimax-Algorithmus als problematisch, da dieser alle Blätter berechnet und bewertet. Dies wiederum benötigt extrem viel Speicher und macht es damit unmöglich, eine große Anzahl von Ebenen zu berechnen. Wie schon zuvor erwähnt, ist es nicht immer sinnvoll, jeden Knoten im Baum zu betrachten. McCarthy (1956) war der Erste, der erkannte, dass Beschneidung beim Minimax-Algorithmus möglich ist. Der Erste jedoch, der eine Lösung dafür zur Verfügung stellte, war Brudno (1963) [**?**]. Der Erfolg der Alpha-Beta-Suche wird dadurch erreicht, dass uninteressante Teil-Bäume abgeschnitten werden.
Beispielsweise max(6, min(5, A)) and min(5, max(6, B)) sind immer gleich 6 und 5, egal welchen Wert die Teilbäume A und B annehmen. Daher können die Teilbäume A und B während des Durchsuchens des Baumes abgeschnitten werden. Um diese Beschneidungsmöglichkeiten zu erkennen, verwendet die Alpha-Beta-Suche ein sogenanntes *Suchfenster* (Alpha, Beta). Dieses wird auf den erwarteten, aktuell betrachteten Wert des Baumes angewandt. Der Wert Alpha bildet dabei die untere, der Wert Beta die obere Schranke. Werte außerhalb dieses Fensters haben keinen Einfluss auf das Ergebnis der Minimax-Suche und müssen daher auch nicht betrachtet werden [**?**]. Die folgende Abbildung zeigt am Minimax-Baum aus Abbildung 5 den Effekt der

Alpha-Beta-Suche:

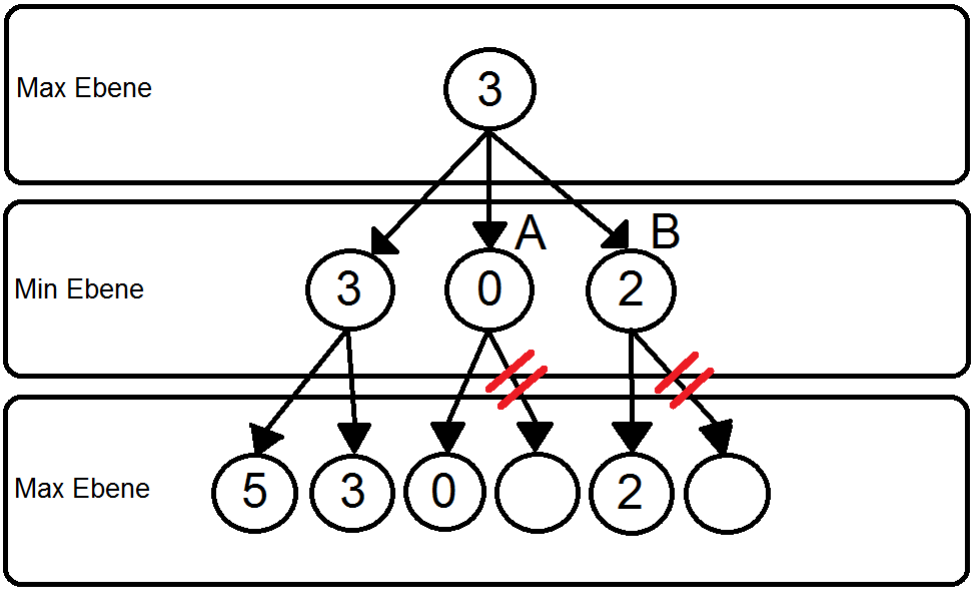

Abbildung 6: Alpha-Beta-Suche auf den Baum aus Abbildung 5 angewendet

Die roten Markierungen in der Abbildung zeigen die Stellen, an denen Teil-Bäume abgeschnitten werden. Nachdem der Wert 3 des linken Teilbaumes bis zur Wurzel hochpropagiert wurde und diese ein Max-Knoten ist, kann das Ergebnis nicht mehr kleiner als 3 werden. Da der hochpropagierte Wert der Knoten A und B jeweils kleiner als diese untere Schranke ist, müssen deren Teil-Bäume nicht mehr betrachtet werden.
Dieses Verfahren eignet sich sehr gut um die Effizienz der Suche in Zwei-Personen-Spielen zu verbessern [**?**, Seite 176] und wird darum auch in dieser Arbeit verwendet.

6 Heuristiken

Das Wort **Heuristik** lässt sich von dem altgriechischen Wort *heurisko* (ich finde) ableiten [Duden]. Bei einer heuristischen Suche handelt es sich um eine Suche, welche auf Vermutungen basiert. Eine Heuristik findet entweder

eine semi-optimale oder keine Lösung, daher werden Heuristiken eingesetzt, wenn keine optimale, aber zumindest eine brauchbare Lösung benötigt wird [**?**]. Man könnte eine Heuristik als das Kernstück eines jeden Agenten betrachten. Im Fall der Brettspiele ist eine Heuristik in der Lage, ein Spielfeld zu bewerten und damit den Wert der Spielsituation einzuschätzen. Diese Einschätzung ist von der Heuristik selbst abhängig. Heuristiken können hier aber auch zur schnelleren Suche im Zustandssuchraum verwendet werden.
George Polya definierte Heuristiken als "das Studium der Methoden und Regeln von Entdeckung und Erfindung" [**?**]. Bei der Zustandsraumsuche werden Heuristiken als Regeln zur Auswahl von Zweigen im Zustandsraum verwendet. Georg F. Luger teilt die Anwendung von Heuristiken bezüglich KI-Problemlösungsverfahren in zwei grundlegende Situationen ein:

1. Ein Problem besitzt keine exakte Lösung, da die Problembeschreibung oder die verfügbaren Daten zweideutig sind. Als Beispiel dafür nennt Luger die medizinische Diagnose. Eine gegebene Menge von Symptomen kann verschiedene Ursachen haben. Ärzte verwenden Heuristiken, um die wahrscheinlichste Diagnose zu wählen und einen Behandlungsplan zu formulieren. [...]

2. Ein Problem besitzt zwar eine exakte Lösung, doch die Rechenkosten für die Suche nach dieser Lösung sind unerschwinglich. Bei vielen Problemen (wie Schach) wächst der Zustandsraum explosionsartig, wobei die Anzahl möglicher Zustände exponentiell oder faktoriell mit der Tiefe der Suche wächst. In solchen Fällen können erschöpfende *Brute-Force*-Suchtechniken wie die Tiefensuche oder die Breitensuche unter Umständen keine Lösung innerhalb eines praktikablen Zeitraums finden. Heuristiken gehen diese Komplexität an, indem sie die Suche entlang des „viel versprechenden" Weges durch den Raum leiten. [...] [**?**]

Doch auch Heuristiken können versagen, denn letztendlich ist eine Heuristik nur eine informierte Vermutung, welcher Schritt als nächstes folgt, um ein Problem zu lösen. Eine Heuristik kann einen Suchalgorithmus zu einer teilweise optimalen Lösung führen oder gar keiner Lösung finden [**?**, Seite 148]. Diese Einschränkung wohnt der heuristischen Suche inne. Sie kann nicht durch „bessere" Heuristiken oder effizientere Suchalgorithmen eliminiert werden [**?**].

6.1 Merkmale starker Heuristiken

Wie bereits erwähnt unterscheiden sich Heuristiken meist, selbst wenn sie das selbe Problem betrachten. Wie sehr sich die Heuristiken unterscheiden, hat mit der Komplexität des Problems zu tun. Je umfangreicher das Problem ist, um so unterschiedlichere Aspekte können in die Bewertung mit einbezogen werden [**?**]. Dennoch lassen sich einige Maße zur Bestimmung der Stärke einer Heuristik formulieren. Dies hat George F. Luger in seinem Buch „Künstliche Intelligenz“ [**?**] getan. Er beschreibt drei allgemeine Maße nach denen sich Heuristiken bewerten lassen:

1. **Zulässigkeit**: Ein heuristische Suche ist *zulässig*, wenn sie garantiert einen minimalen Weg zu einer Lösung findet, falls ein solcher Weg existiert. Zum Beispiel ist die Breitensuche eine zulässige Suchstrategie, da sie jeden Zustand einer Ebene n des Graphen untersucht, bevor sie zu Ebene $n + 1$ wechselt. Dadurch werden alle Zielknoten entlang des kürzestmöglichen Wegs gefunden.

2. **Monotonie**: Eine heuristische Funktion ist *monoton*, wenn sie „lokal zulässig“ ist, also konsistent den minimalen Weg zu jedem Zustand findet, auf den sie während der Suche trifft.

3. **Informiertheit**: Eine Heuristik zum selben Problem ist nach Luger „besser“ als eine andere, wenn sie *informierter* ist. Das bedeutet, dass die *informiertere* Heuristik mehr Expertenwissen über des zu bearbeitende Problem besitzt und somit in weniger Zügen zu einem Gewinnbringenden Zustand kommt.

An Hand dieser Kriterien soll nach Möglichkeit die Stärke der implementierten Heuristiken eingeschätzt werden. Falls nicht alle Kriterien anwendbar sind, werden weitere Überlegungen zur Heuristik vorgenommen.

7 Technische Umsetzung

Nachdem in den vorherigen Kapiteln die Thematik der Arbeit beschrieben wurde, beschäftigen sich die folgenden Kapitel mit der Umsetzung der Verfahren und den dafür entwickelten Heuristiken. Zuerst wird jedoch der Gesamtaufbau des entwickelten Programms beschrieben.

7.1 Die Software

Programmiert wurde die Software in JavaSE-1.6. Java wurde gewählt, da es durch die Verwendung von Bytecode plattformunabhängig ist und auch Vererbung unterstützt. Die Vererbung wird benötigt um verschiedene Spieler mittel eines Interfaces gegeneinander spielen lassen zu können. Der Garbage Collector bietet außerdem eine automatische Speicher- und Heap-Verwaltung, welche beim häufigen Erstellen und Verwerfen der Spielbäume benötigt wird. Ein weiterer Grund für die Wahl von Java ist das einfache Strukturieren mittels Paketen, welche hier nach der *MVC-Struktur* (Model-View-Control) eingeteilt wurden[7]. Das wichtigste Merkmal dieser Struktur ist die genaue Unterteilung von Aufgabenbereichen innerhalb der Software. Grafisch sieht die Funktionsweise der Software folgendermaßen aus:

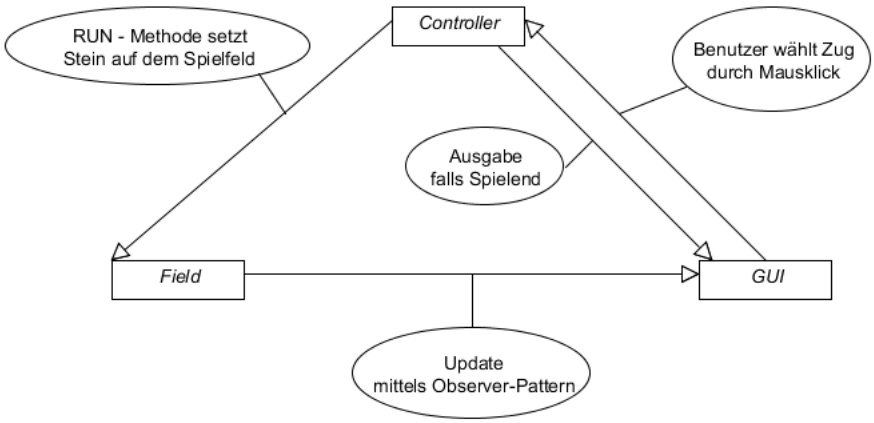

Abbildung 7: Funktionsweise der MVC-Struktur

Zusätzlich zur MVC-Struktur wurde noch ein Paket hinzugefügt, welches die Implementierung der Spieler umfasst. Dieses beinhaltet Klassen für den *menschlichen Spieler* (HumanPlayer), einen *zufällig* spielenden Agenten, einen Agenten, welcher *Influence Mapping* verwendet (InfMapKI), einen Agenten, welcher einen *Spielbaum* verwendet (SearchTreeKI) und ein Interface, welches es ermöglicht, die genannten Spieler in verschiedener Kombination gegeneinander Spielen zu lassen. Beim Start der Software wird ein

[7]Ein Diagram zur Übersicht findet sich im Anhang

neuer Controller erstellt und dieser wird mit einem Thread gestartet. Die Größe des Spielbretts lässt sich in der Klasse Field anpassen. Für die implementierten Agenten ist die Größe des Spielfeldes irrelevant. Die Funktionen sind derart programmiert, dass sie sich auf jede Brettgröße anwenden lassen. Diese Funktionalität erleichtert den Vergleich mit anderen Agenten, da diese sehr unterschiedliche Feldgrößen benutzen. Außerdem ist es damit möglich, die Funktionen der Agenten auf kleinen Feldern mit beispielsweise 8x8 Positionen zu testen. Ein kleiners Feld hat beim Spielbaum den Vorteil, dass der Baum nicht so schnell wächst.

7.1.1 Die Steuerung

Das Paket, welches die Steuerung der Software umfasst, beinhaltet einen *Controller* (GobangController), welcher auch die Regeln für Gomoku beinhaltet, zwei Klassen für die Umsetzung der Regeln von Pente und Gobang, so wie einen Clicklistener zur Steuerung durch menschliche Spieler.
Der **Controller** ist für die Steuerung des Spiels zuständig und enthält die gesamte Spiellogik für das Spiel Gomoku, da dies das einfachste der drei Spiele ist. Bei Bedarf kann auf die Regelerweiterungen der anderen beiden Spiele zugegriffen werden, welche sich im selben Paket befinden. Um zu verhindern, dass ein Spieler einen Spielstein setzt, während der Andere am Zug ist, stellt der Controller ein Lock-Objekt zur Verfügung, welches dem menschlichen Spieler nur übergeben wird, falls er selbst am Zug ist. Bei seiner Initialisierung erstellt der Controller außerdem ein neues Spielbrett und eine neue Benutzeroberfläche, an welche das neue Feld und das Lock-Objekt übergeben werden. Der Controller selbst implementiert das Java-Interface *Runnable* und der Ablauf eines Spiels wird daher in einer Run-Methode geregelt.
Während eines laufenden Spiels wartet der Controller abwechselnd auf die Züge der beiden Spieler. Ist ein Zug gültig, so wird dieser auf dem Spielbrett gesetzt und die gewählten Spielregeln werden angewandt. Nach einem solchen Zug wird mittels der *endGame*-Methode überprüft, ob einer der beiden Spieler gewonnen hat oder das Spielbrett vollständig mit Spielsteinen belegt ist. Sollte dies der Fall sein, so wird das laufende Spiel beendet und es kann ein neues gestartet werden. Ist ein Zug ungültig, so wartet der Controller kurz und fordert danach einen neuen Zug des Spielers an. In der *startGame*-Methode kann festgelegt werden, mit welchen Parametern der Agent aufgerufen wird, der den Spielbaum verwendet. Weitere Informationen dazu befinden sich in Kapitel 7.3. Zur Illustration werden nun die Klassen des Paketes *Control* in der UML-Darstellung aufgelistet:

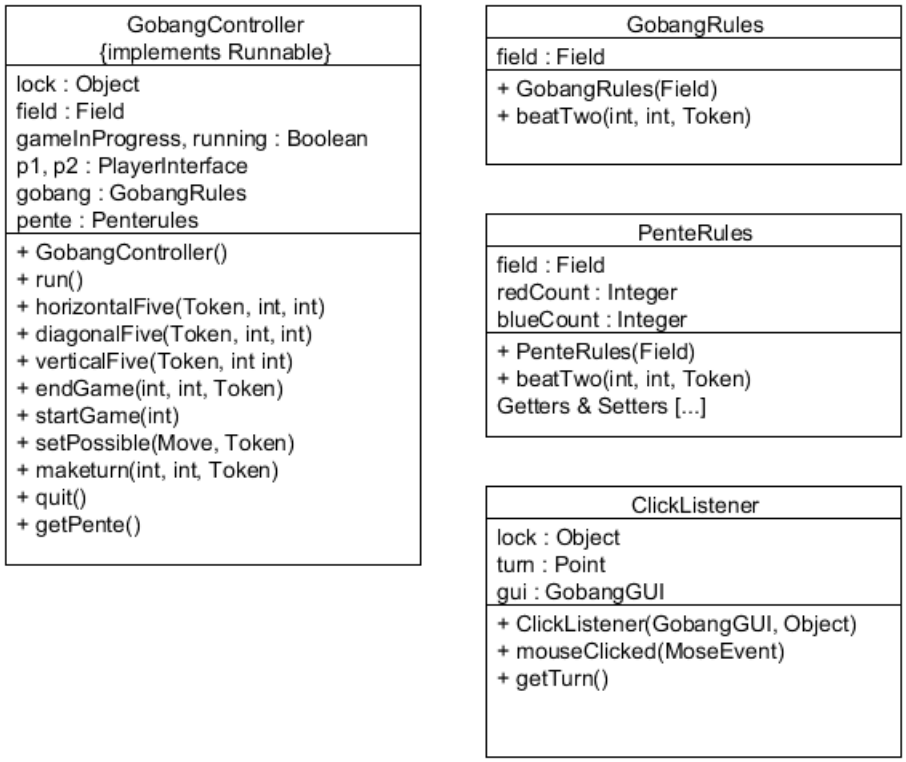

Abbildung 8: UML-Klassendiagramm: Control

7.1.2 Das Modell

Im Paket für das Modell befinden sich die Datenstrukturen für das *Spielbrett* (Field) und ein dazugehöriges Interface (Modelinterfance), den *Zug* (Move), einen Datentyp für die *Spielsteine* (Token) und je einen Datentyp für die jeweiligen Bedürfnisse der beiden Ki-Verfahren. Letztere werden in den Kapiteln zur Umsetzung der KI-Verfahren erläutert.
Das **Spielbrett** wird durch ein zweidimensionales Array dargestellt und erweitert das Java-Interface *Observable*, um Änderungen am Array an die beobachtenden Klassen weitergeben zu können. Jede Position des Arrays enthält einen Spielstein. Die verschiedenen **Spielsteine** sind als Enumeration in der Klasse Token bespeichert. Bei der Initialisierung eines neuen Spielbrettes werden alle Positionen des Arrays mit dem Spielstein *EMPTY* beleget, welcher die freie Position repräsentiert. Ein **Zug** wird durch einen Point mit den gewählten Zielkoordinaten dargestellt. Führt ein Spieler einen solchen Zug aus und handelt es sich um einen legalen Zug, so wird der Spielstein auf der

im Zug angegebenen Position durch einen Spielstein des ziehenden Spielers ersetzt. Abschließend werden noch einmal die Klassen des Paketes Model in der UML-Darstellung aufgeführt:

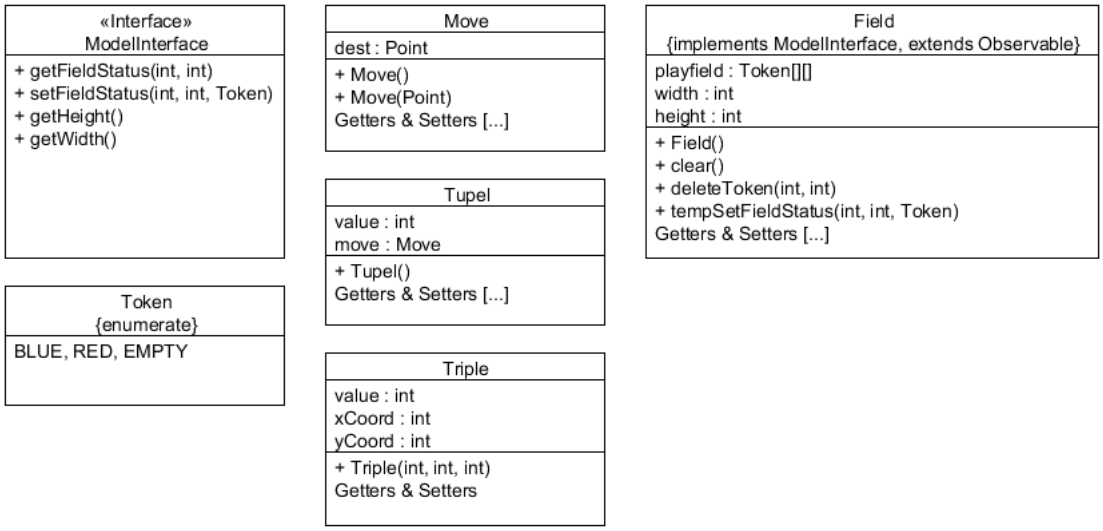

Abbildung 9: UML-Klassendiagramm: Model

7.1.3 Die Ansicht

Das Paket für die Ansicht der Software umfasst alle Klassen, die zur Darstellung des Spiels und des Spielfeldes benötigt werden. Dazu gehören die *Benutzeroberfläche* (GobangGUI), die *Darstellung des Spielfeldes* (PaintGobang) und ein Startlistener zum Initialisieren eines *Lock*-Objekts, welches wiederum zur Steuerung der Nebenläufigkeit dient.
Die **Benutzeroberfläche** dient überwiegend zu Testzwecken und ist daher sehr schlicht gehalten. Sie besteht aus einem Fenster, welches sich an die Größe des Bildschirms anpassen lässt und verfügt über ein Drop-Down-Menü zur Auswahl der Spieler, ein weiteres zur Auswahl des zu spielenden Spieles und einen Startknopf. Des Weiteren zeigt ein Label am unteren Rand des Fensters an, welcher Spieler gerade am Zug ist und falls es sich bei dem gewählten Spiel um Pente handelt, werden zwei weitere Labels hinzugefügt, welche die Anzahl der bereits geschlagegen Spielsteine anzeigen. Die Benutzeroberfläche selbst wird durch den Controller aufgerufen und gibt die durch den Benutzer gewählten Menüeinstellungen an diesen zurück. Außerdem implementiert sie das Java-Interface *Observer* und zeichnet das Fenster neu,

falls sich Änderungen am Modell ergeben.
Die **Darstellung des Spielbretts** nimmt den Großteil des durch die Benutzeroberfläche generierten Fensters ein. Diese Klasse zeichnet abhängig von der Größe des gesamten Fensters eine schlichte, aber deutliche Darstellung des Spielbretts. Die Spielsteine des beginnenden Spielers werden als rote, die des zweiten Spielers als blaue Punkte dargestellt. Auf dem folgenden Bild ist ein Beispielzustand der Benutzeroberfläche inklusive Darstellung des Spielbretts zu sehen. Der grün eingerahmte Bereich ist dabei die Benutzeroberfläche, der gelb eingerahmte Bereich die Darstellung des Spielbretts.

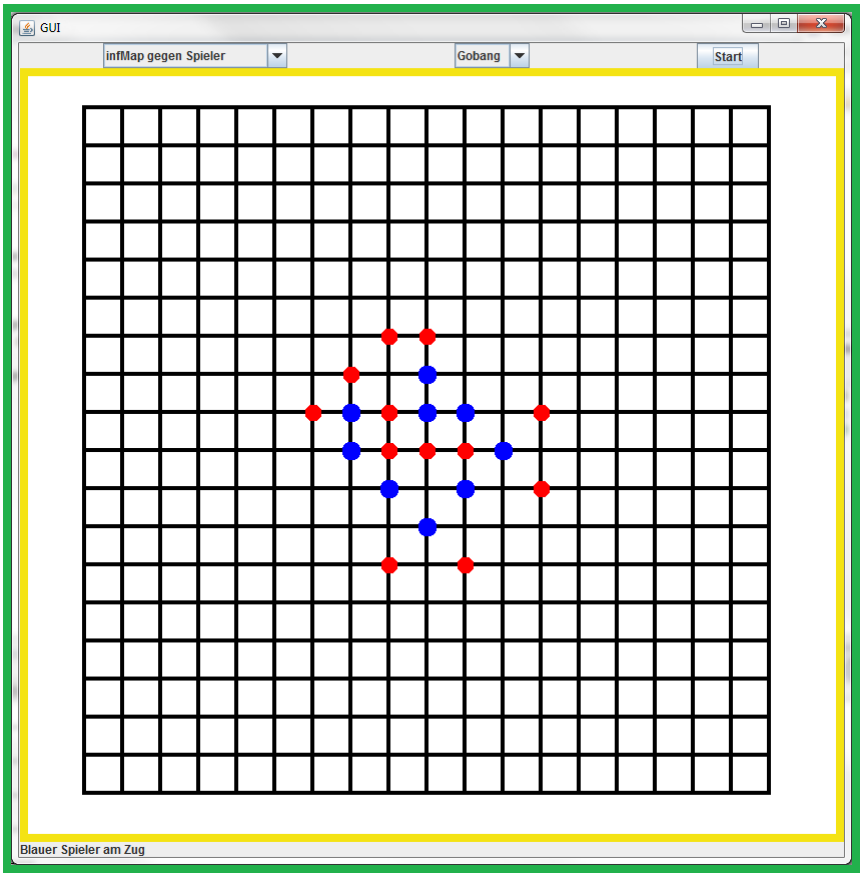

Abbildung 10: Benutzeroberfläche

Die Klassen des Paketes *View* werden im Folgenden noch einmal in der UML-Darstellung aufgeführt:

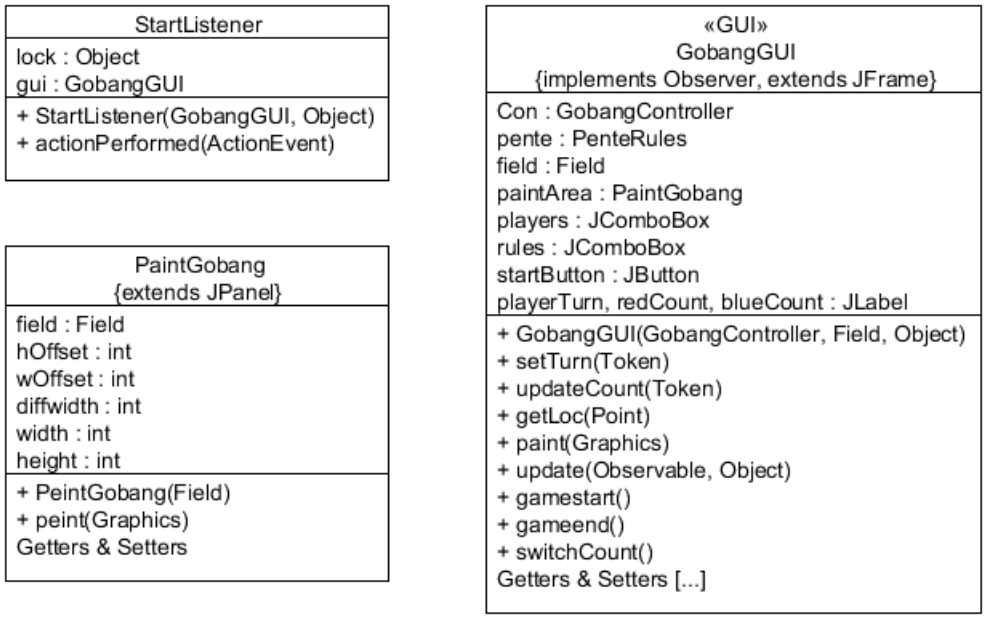

Abbildung 11: UML-Klassendiagramm: View

7.2 Die Influence Map als Kern des Agenten

Zum Vergleich zweier verschiedener Verfahren der Zuggenerierung für Brettspiel-Agenten wird zuerst der Agent betrachtet, welcher das Influence Mapping und eine dazu passend entwickelte Heuristik verwendet. Von diesem Verfahren werden eine schnelle Zugwahl, sowie ein geringer Speicherbedarf im Vergleich zum Spielbaum-Agenten erwartet. Da das Verfahren in der hier verwendeten Form praktisch gesehen nur einen Zug vorausschaut, muss die hier verwendete Heuristik deutlich informierter als die des anderen Agenten sein, um die selbe Spielstärke zu erzielen. Dafür steht dem Agenten aber auch verhältnismäßig viel Zeit zur Verfügung, um das einzelne Feld zu analysieren. Der andere Agent muss in dieser Zeit viele Felder analysieren. Im Folgenden wird die Funktionsweise des Agenten so wie die verwendete Heuristik beschrieben.

7.2.1 Funktionsweise

Bei der Initialisierung durch den Controller bekommt der Agent die Benutzeroberfläche übergeben und kann mittels dieser die Maße des Spielbretts

auslesen. Mit diesen Maßen wird dann eine neue **Influence Map** in der Größe des Spielbretts erstellt.
Obwohl Influence Maps auch sehr viel mehr Daten enthalten können, wird hier zur Umsetzung eine schlichte Variante verwendet. Diese wird analog zum Spielbrett durch ein zweidimensionales Array dargestellt. Dieses Array wird jedoch nicht mit Spielsteinen, sondern mit Integer-Werten beschrieben. In diesem Array wird beim Erstellen jede Position mit dem Wert 100 belegt. Ist eine Position mit einem größeren Wert als Null belegt, so ist das Setzen von Spielsteinen an der entsprechenden Position auf dem Spielfeld erlaubt.
Um einen **Zug** zu bestimmen wird die Influence Map ähnlich einer Landkarte zu Rate gezogen um die beste Position, ähnlich einer besten Route, zu finden. Der Wert jeder Position der Map wird dafür mit dem höhsten Wert der bisher betrachteten Positionen vergleichen. Ist der Wert höher als alle bis dahin betrachteten Werte, so wird diese Position als nächster Zug gespeichert. Sind die Werte der zwei verglichenen Positionen gleich groß, so wird durch Zufall eine der beiden Positionen als aktuell beste gespeichert. Diese einfache Vorgehensweise ermöglicht es, dass der Agent nicht jedes mal gleich agiert, selbst wenn sein Gegenspieler die selben Züge macht.
Vor jedem Zug des Agenten, wird die Map aktualisiert, indem die ensprechenden heuristischen Funktionen auf das Spielfeld angewandt werden. Ist eine Position auf dem Spielfeld belegt, so wird sie mit Null bewertet und steht als Zug nicht mehr zur Verfügung. Ist noch kein Spielstein auf dem Brett, so wird die Mitte des Bretts für den ersten Zug gewählt, da gewinnbringende Spielstellungen in *Verbinde k*-Spielen häufig einen Stein in Feldmitte beinhalten[**?**]. Am Ende wird der Zug an den Controller zurückgegeben, welcher die Position auf der Influence Map repräsentiert die am höchsten bewertet wurde. Es wird in dieser Implementierung davon abgesehen eine zweite Influence Map für die Züge des Gegners zu generieren, da die im Folgenden erklärte Heuristik die Bedrohung durch den Gegner schon in die Bewertung der Positionen mit einfließen lässt.
Da unter Windows ein *Zugriffverletzungs*-Fehler[8] auftritt, wenn Fünfer-Reihen erkannt werden, musste für diesen Fall noch eine Fehlerbehebung entwickelt werden. Die Funktion *updateInfMap*, welche vor jedem eigenen Zug zum Aktualisieren der Influence Map aufgerufen wird, gibt nun eine Liste von *Tripeln* zurück. Diese Liste beinhaltet alle Positionen auf dem Spielfeld, welche als Ende einer Vierer-Reihe erkannt wurden und ihren dazugehörigen Wert. Die Daten werden alle als Integer im Datentyp Tripel gespeichert und bilden einen Eintrag der Liste. Nach dem eigentlichen Aktualisieren der Influence Map werden diese Einträge noch einmal der Influence Map hinzugefügt. Da-

[8] ExceptionCode=0xc0000005

mit lässt sich der Zugriffsfehler vermeiden.

7.2.2 Heuristik der Mustererkennung

Für die meisten Spiele gibt es nachezu unendlich viele Möglichkeiten, Heuristiken zu entwerfen [**?**]. Allerding sollte die Heuristik das verwendete Verfahren zur Bestimmung der Züge unterstützen. Die implementierte Heuristik legt ihren Schwerpunkt in Gomoku ausschließlich auf das Angreifen durch eigene Reihen und das Blockieren gegnerischer. Wird Gobang oder Pente gespielt, werden noch weitere Bewertungsfunktionen verwendet und der Agent legt eine höhere Priorität darauf, Zweier-Reihen des Gegners zu schlagen.
Die beim Influence Mapping verwendete Heuristik wertet bezüglich jeder Zelle ihre Eigenschaft auf das Spiel aus Sicht des Agenten aus. Alle Eigenschaften haben eine eigene Gewichtung und nach der Auswertung erhält die Zelle das Gewicht der wichtigsten Eigenschaft. Um dieses zu ermitteln wird vor dem Wählen eines Zuges die Influence Map mittels der *updateInfMap*-Funktion aktualisiert. Die Funktion geht jede Position des Spielbretts durch und ruft auf jeder dieser Positionen alle **Funktionen zur Mustererkennung** auf. Diese Funktionen werden in einer **Prioritätsreihenfolge** aufgerufen, welche aufsteigend vom niedrigst bewerteten zum höchst bewerteten Muster sortiert ist. So wird gewährleistet, dass nach dem aktualisieren der Influence Map die höchsten Werte in den jeweiligen Zellen stehen. Die Werte der Influence Map werden allerdings nur geändert, falls die betreffender Zelle nicht zuvor mit Null bewertet wurde, da diese Zellen auf dem Spielbrett ja bereits belegt sind.
Man könnte argumentieren, dass der Aufruf aller Funktionen zur Mustererkennung auf jeder Position des Spielbretts unnötig viel Zeit in Anspruch nimmt. Daher sind diese Funktionen derart gestaltet, dass Abfragen gar nicht erst durchgeführt werden, falls die zu überprüfenden Positionen außerhalb des Feldes liegen. Für alle Positionen in der Nähe des Randes des Spielbretts werden also insgesamt deutlich weniger Funktionen aufgerufen. Damit wird auch direkt sicher gestellt, dass keine ArrayOutOfBounds-Exceptions beim Prüfen der beteiligten Positionen auftreten, falls diese außerhalb des Spielbretts liegen. Im folgenden Auszug aus der Software ist dies deutlich an den If-Schachtelungen zu erkennen.

Listing 3: Schachtlung einer Mustererkennungsfunktion

```
1  private void findTwo(Field fld, Token tok, int column, int row){
       int width =
2  fld.getWidth(); int height = fld.getHeight();
3    int value;
4
5    //how to rate
6    if(tok == getColor()){
7      value = 400;
8    } else value = 500;
9
10   if(fld.getFieldStatus(column, row) == tok){
11
12     //vertical
13     if(row == 0){
14       if(tok == getColor() && fld.getFieldStatus(column, row+1)
           == tok){
15         changeInfMap(column, row+2, value);
16       }
17
18     } else if(row == height -1){
19       if(tok == getColor() && fld.getFieldStatus(column, row-1)
           == tok){
20         changeInfMap(column, row-2, value);
21       }
22   [...]
23 }
```

Jede dieser Funktionen ist so implementiert, dass man ihr eine Spielerfarbe übergeben kann. Mittels dieser wird das jeweilige Muster entweder für den Agenten selbst oder seinen Gegenspieler gesucht. Abhängig von der übergebenen Farbe wird auch die Position der Influence Map bewertet. Die Zeilen sind im Beispielcode mit „how to rate“ kommentiert.
Die Heuristik ist in der Lage, einfache Reihen wie auch komplexere Spielstellungen zu erkennen, wie in folgender Tabelle aufgelistet:

Eigenschaft	Bewertung	Gobang/Pente spezifisch
Feld belegt	0	
Möglicher Zug	100	
Eigene Steine gefährden	100	X
Neben gegnerischem Stein	200	
Neben eigenem Stein	300	
Ende eigener Zweier-Reihe	400	
Ende einseitig geblockter Dreier-Reihe	400	
Lücke zwischen gegnerischen Steinen	450	
Lücke zwischen eigenen Steinen	475	
Angriff zweier gegnerischer Steine	500	X
Angriff beenden	600	X
Gegnerischen Angriff verhindern	700	X
Doppel-Angriff	750	X
„Dread X“	700	
Gegnerische Doppel-Drohung	775	
Eigene Doppel-Drohung	780	
Unblockierte gegnerische Dreier-Reihe	800	
Unblockierte eigene Dreier-Reihe	900	
Gegnerische Vierer-Reihe blockieren	1000	
Eigene Vierer-Reihe beenden	1100	

Die in der Tabelle als **Gobang/Pente spezifisch** markierten Funktionen beziehen sich auf das Schlagen von gegnerischen und das Verteidigen eigener Spielsteine.
Falls die zu bewertende Zelle nicht als gewinnbringend[9] oder niederlagenverhindernd[10] eingestuft wird, addiert die Heuristik noch einen Wert zum Wert der Zelle, welcher höher ausfällt, je näher sich die Zelle am Mittelpunkt des Spielfeldes befindet. Damit wird gewährleistet, dass Zellen in der Feldmitte bevorzugt werden. Sowohl Spalten als auch Zeilen werden bei dieser Rechnung berücksichtigt. Dies ist sinnvoll, da die meisten gewinnbringenden Stellungen in *Verbinde k*-Spielen Steine in der Feldmitte beinhalten [**?**]. Die Addition dieses Wertes im Bereicht von 0 bis 18 ist der Grund für die Bewertung in 50er- und 100er-Schritten. Bei diesen Werten entscheidet der aufaddierte Wert nur welche der Positionen bei gleichem Wert gewählt wird. In der Tabelle findet sich ein Fall in dem der Bewertungsunterschied zweier Muster kleiner als 18 ist[11]. In diesem Fall soll der Agent sich für die nor-

[9] bezüglich des nächsten Zuges

[10] eben dies

[11] Gegnerische Doppel-Drohung und Eigene Doppel-Drohung

malerweise geringer bewertete Position entscheiden können, falls sie deutlich näher an der Spielbrettmitte liegt. Bei diesem Muster handelt es sich um einen Angriffszug, welcher zwei Reihen gleichzeitig bedroht. Es besteht also eine 100% Chance, zwei Steine des Gegners zu entfernen. Diese soll nahe der Spielbrettmitte genutzt werden, um Platz für eingene Steine zu schaffen.
Für einen gegebenen Feldzustand (links) sieht die Influence Map des roten Spielers demnach folgendermaßen aus (rechts): je dunkler die Position markiert ist, um so interessanter ist sie für den roten Spieler.

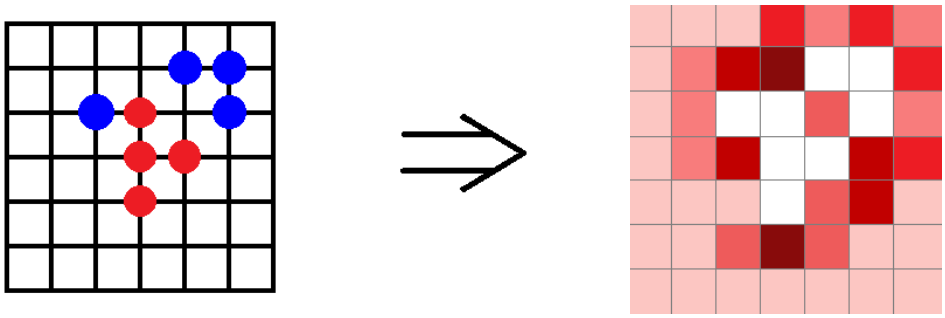

Abbildung 12: Beispiel: Influence Map zum Spielbrett

Bei der Betrachtung der Influence Map ist zu erkennen, dass diese Heuristik vor allem die Endpunkte von zusammenhängenden Reihen einer Farbe markiert. Die bereits auf dem Spielbrett belegten Positionen sind weiß markiert.
Die Bewertung der Muster verfolgt bis auf zwei Fälle eine defensive Strategie, da diese gegen ausschließlich offensiv ausgelegte Spieler häufig gewinnt[12] [13]. Die zwei Fälle sind das Erweitern einer beidseitig offenen Dreier-Reihe und das Vervollständigen einer Fünfer-Reihe. Diese werden höher bewertet als Reihen des Gegners mit diesen Eigenschaften, da sie unvermeidbar zum Sieg führen. In allen anderen Fällen werden Züge, die das Geschlagen werden eigener Steine oder das Ausbauen von gegnerischen Reihen verhindern höher bewertet als solche, die bei einem Angriff auf gegnerische Steine beteiligt sind.
Bezüglich der drei von Luger [**?**] vorgestellten Kriterien zur Bewertung einer Heuristik eignet sich bei dieser Technik nur eins. Da sowohl Monotonie als auch Zulässigkeit sich auf die Kürze der *Wegfindung* in Bäumen beziehen, können sie für das Influence Mapping nicht überprüft werden, da theoretisch

[12] pente.org(29.8.2012)

[13] http://www.generation5.org/content/2001/penteai.asp (14.7.2012)

nur eine Ebene im Voraus brachtet wird. Zur **Informiertheit** lassen sich allerdings einige Aussagen tätigen:
Vergleicht man die Anzahl der Muster[14], welche von dieser Heuristik erkannt werden, so übersteigt diese deutlich die Anzahl[15] der von James Methews[16] vorgeschlagenen Funktionen für einen Agenten der das selbe Problem lösen soll. Außerdem bestimmt die *Heuristik der Mustererkennung* für jede Position auf dem Spielbrett einen Wert, während viele Heuristiken die einen Spielbaum verwenden nur den Zustand des gesamten Spielbretts bewerten. Bezüglich der *Informiertheit* wird noch angewerkt, dass diese immer noch viel Verbesserungspotential bietet, in dem noch weitere Muster hinzugefügt werden.

7.2.3 Vor- und Nachteile

Ein entscheidener Nachteil diser Umsetzung ist die Abhängigkeit von einer guten Heuristik. Die Spielstärke des Agenten ist allein von der Heuristik abhängig, da sonst keine Techniken wie zum Beispiel die Propagierung von Werten oder eine zweite Influence Map für den Gegenspieler verwendet werden.
Ein weiterer Nachteil der Anwendung von Influence Mapping auf die umgesetzten Brettspiele ist, dass ein Propagieren der Werte auf der Influence Map keinen Sinn macht und der Agent daher nur einen Zug im Voraus planen kann.
Ein Vorteil des Influence Mappings ist seine Geschwindigkeit. Im direkten Vergleich mit dem Spielbaum berechnet der Agent seinen Zug 300 Mal so schnell wie sein Gegenspieler[17].
Ein weiterer Vorteil ist die Nachvollziehbarkeit der ausgewählten Züge. Durch eine einfache Konsolenausgabe des Zuges mit Wert, kann man leicht überprüfen ob dieser richtig bestimmt wurde.
Der wahrscheinlich größte Vorteil dieser Umsetzung ist die große Flexibilität. Das Verfahren des Influence Mapping ist von sich aus schon sehr flexibel und kann auf die unterschiedlichsten Probleme der Spieleentwicklung angewandt werden[**?**]. Dazu passend können der hier verwendeten Heuristik weitere Bewertungsfunktionen für komplexere Muster hinzugefügt werden, wenn diese an der passenden Stelle in die *updateInfmap*-Funktion eingefügt werden.

[14] 17 verschiedene Muster

[15] 6 verschiedene Muster

[16] http://www.generation5.org/content/2000/boardai.asp

[17] siehe Kapitel 8.2

Listing 4: updateInfmap

```
1  private LinkedList<Triple> updateInfMap(Field fld){
2    int width = fld.getWidth();
3    int height = fld.getHeight();
4    LinkedList<Triple> fourRow = new LinkedList<Triple>();
5
6    for (int i = 0; i < width; i++) {
7      for (int j = 0; j < height; j++) {
8          //heuristics with priority - least first
9
10       //updates blocked positions
11       if(fld.getFieldStatus(i, j) != Token.EMPTY){
12                 influenceMap[i][j] = 0;
13       }
14
15       findSingle(fld, getOpponentsColor(), i, j);
16       [...]
17
18           //Gobang and pente rules
19           if(gui.getRules() > 0){
20
21             findTwo(fld, getOpponentsColor(), i, j);
22
23             possibleComplete(fld, getColor(), i, j);
24
25             possibleComplete(fld, getOpponentsColor(), i, j);
26
27             //advanced gobang and pente moves
28                 selfTrap(fld, i, j);
29
30                 doubleAttack(fld, i, j);
31           }
32
```

```
33          dreadX(fld, i, j);
34
35          doubleThreat(fld, getOpponentsColor(), i, j);
36              [...]
37          }
38      }
39      return fourRow;
40  }
```

In diesem Code-Abschnitt erkennt man, dass unter „advanced gobang and pente moves“ die zwei Muster *selfTrap* und *doubleAttack* überprüft werden. Diese wurden im Nachhinein hinzugefügt um die Informiertheit der Heuristik zu verbessern.

7.3 Der Spielbaum als Kern des Agenten

Nach der zu Beginn des Kapitels vorgestellten Einteilung von Luger [**?**] fällt dieses heuristische Verfahren in die zweite Kategorie. Das Verfahren ermöglicht in diesem Fall die Pfadsuche im Spielbaum (dem Suchraum). Die vom Negamax-Algorithmus berechneten Knoten werden durch die Heuristik ausgewertet und dann mit dem Alpha-Beta-Schranken geprüft, um zu entscheiden, ob der Teil-Baum noch weiter betrachtet werden muss oder abgeschnitten werden kann. Die Heuristik bewertet dafür das zu einem Knoten gehörende Spielbrett und gibt diesem einen Wert. Aufgrund der vielen zu prüfenden Knoten wird eine weniger umfangreiche Heuristik als beim anderen Agenten verwendet.

7.3.1 Funktionsweise

Bei der Initialisierung durch den Controller bekommt der Agent die Benutzeroberfläche so wie eine zuvor festgelegte maximale Suchtiefe und einen Radius übergeben, welcher bei der Generierung des Baumes verwendet wird. Die Generierung des Baumes und die gleichzeitige Suche nach dem bestmöglichen Zug werden durch den in Kapitel 5.2.1 beschriebenen Negamax-Algorithmus durchgeführt. Die Implementierung ist im Folgenden zu sehen.

Listing 5: Negamax-Algorithmus

```
1  private Tupel negaMax(Field fld, int depth, int alpha, int beta,
       boolean kiTurn){
2    Tupel t = new Tupel();
3    if(depth == 0 || gameEnd(fld)){
4      t.setValue(-evaluate(fld));
5      return t;
6    }
7    ArrayList<Move> moves = generateMoves(fld, kiTurn);
8    t.setValue(-Integer.MAX_VALUE);
9
10   for(Move m : moves){
11     fld.tempSetFieldStatus(m.getDest().x, m.getDest().y, kiTurn?
           getColor() :
12 getOpponentsColor()); int value = -negaMax(fld, depth-1, -beta,
     -alpha, !kiTurn).getValue();
13
14     fld.tempSetFieldStatus(m.getDest().x, m.getDest().y, Token.
         EMPTY);
15     if(value > t.getValue()){
16       if(value > alpha){
17         alpha = value;
18       }
19       t.setMove(m);
20       t.setValue(value);
21       if(alpha >= beta){
22         break;
23       }
24     }
25   }
26   return t;
27 }
```

Erläuterung der Funktionsweise: Zuerst erstellt der Negamax-Algorithmus ein neues Tupel *t*. Der Datentyp Tupel enthält einen Zug und dessen Bewer-

tung. Dieses Tupel t wird nach dem Ausführen des Algorithmus zurückgegeben. Die Zeilen 3-6 legen das Abbruchskriterium für die folgende Rekursion fest. Ist die maximale Suchtiefe erreicht so wird der Knoten im negierten Sinn durch Negamax von der Heuristik ausgewertet und das Tupel t erhält den Wert. Die Funktion gameEnd stellt eine der Beschränkungen des Baumes dar. Die Funktion prüft, ob einer der beiden Spieler bereits gewonnen hat. In diesem Fall wird der Baum von diesem Knoten aus nicht weiter vertieft und dieser Knoten wird ebenfalls durch die Heuristik bewertet.
Tritt keines der Abbruchkriterien auf, so werden mittels der Funktion **generateMove** die möglichen Folgezüge generiert. GenerateMove bekommt das Feld übergeben, welches als Grundlage für die zu generierenden Züge dient. Um dem Problem des zu großen Suchraums entgegenzuwirken, wird hier ein selbst entwickeltes Konzept verwendet, welches sich **Radiusgenerierung** nennt. Es ist relativ simpel, doch handelt es sich dabei um die effektivste Beschränkung des Baumes, welche von der Software verwendet wird. Die Idee liegt darin, nur Züge zu generieren, welche sich in einem bestimmten Radius um die auf dem Feld existierenden Spielsteine befinden. Dieser Radius wird dem Agenten bei der Initialisierung, genau wie die maximale Suchtiefe, vom Controller übergeben. Mit diesen beiden Parametern lassen sich die Spielstärke, sowie Zeit- und Speicherbedarf des Agenten regulieren. Um keine Züge außerhalb des Spielbretts zu generieren, wird zuerst von einer Hilfsfunktion der Abstand zu den jeweiligen Rändern des Spielbretts bestimmt. Ist der Radius größer als einer dieser Abstände, wird beim Generieren der Abstand statt des Radius bezüglich dieser Richtung verwendet. Befindet sich der Spielstein beispielsweise direkt am oberen Rand des Spielbretts, so werden nur Züge in einem Viereck links, rechts und unter dem Spielstein im Bereich des Radius generiert, wie im folgenden Beispiel orange, für den Radius mit Länge 1, markiert.

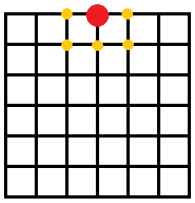

Abbildung 13: Radiusgenerierung

Die Radiusgenerierung wird für *alle* auf dem Spielfeld gesetzten Steine durchgeführt. Da sich die Radien mehrerer Spielsteine überschneiden können, wird für eine bestimmte Koordinate nur ein Zug den Folgezügen des Knotens

hinzugefügt. Damit wird die doppelte Betrachtung von Folgezügen verhindert, was dem Baum weiter einschränkt. Werden Folgezüge auf diese Weise generiert, schränkt man damit den Verzweigungsgrad für große Spielbretter enorm ein. Für ein Go-Brett liegt der Verzweigungsgrad, falls sich ein Spielstein auf dem Spielbrett befindet, in der ersten Baumebene bei 370, in der Zweiten bei 369 usw. Verwendet man die Radiusgenerierung mit dem Radius 2 und liegt der Stein weit entfernt vom Rand, so beträgt der verzweigungsgrad in der ersten Baumebene 24 und in der zweiten maximal 40. Nach der von Quian getätigten Abschätzung der Knotenanzahl von $Verzweigungsgrad^{AnzahlderEbenen}$ [**?**] lässt sich an einem Beispiel verdeutlichen, welchen Effekt die Radiusgenerierung hat. Für die Radiusgenerierung wird im Beispiel der Radius 2 verwendet und es wird der „worst Case“ betrachtet. Dieser tritt auf, falls die gesetzten Spielsteine den diagonal maximalen Abstand zueinander haben, denn dann werden am wenigsten doppelte Kinder aussortiert. Auf dem Go-Brett liegt ein einzelner Spielstein im Zentrum.

verwendete Technik	Ebenen	geschätzte Knotenanzahl
Go-Brett ohne Einschränkung	1	370
Radiusgenerierung	1	24
Go-Brett ohne Einschränkung	3	46.268.000
Radiusgenerieurung	3	4.096
Go-Brett ohne Einschränkung	5	5.880.511.900.000
Radiusgenerieurung	5	1.048.000

Mit dieser Technik lässt sich also nicht das Wachsen des Verzweigungsgrades verhindern, allerdings wird er starkt eingeschränkt, wodurch weniger Zeit und Speicher benötigt wird und damit das Generieren weiterer Ebenen möglich ist.
Als Gegenargument für die Radiusgenerierung könnte man anführen, dass eventuell wichtige oder sogar spielentscheidene Züge nicht betrachtet werden, da sie möglicherweise nicht innerhalb des Radius liegen. Zur Erkennung einer Reihe, welche umgehend blockiert werden muss, reicht allerdings schon ein Radius der Länge zwei aus. Zu diesen Reihen gehören die beidseitig offene Vierer-Reihe und die Fünfer-Reihe. Um diese in zwei Zügen zu generieren wird eine Dreier-Reihe benötigt. Zum Generieren der Füfer-Reihe muss geprüft werden, ob an einem Ende der Reihe zwei Positionen frei sind. Für eine offene Vierer-Reihe muss das selbe auf beiden Seiten der Dreier-Reihe geprüft werden. In beiden Fällen reicht offensichtlich ein Radius der Länge zwei aus. Die Technik der Radiusgenerierung ist also unproblematisch.
In Zeile 8 des Negamax-Algorithmus wird der Wert des Tupels maximal gering gewählt, welches eine Vorbereitung für die Alpha-Beta-Suche ist. Die

folgende Schleife stellt das Kernstück des Algorithmus dar. Sie wird für jeden zuvor mittels generateMove generierten Zug durchgeführt. Der übergebene Wert *kiTurn* bestimmt, ob die zu generierenden Züge von dem Agenten (der KI) oder dem Gegenspieler ausgeführt werden, also in welcher Farbe die Züge gesetzt werden. Zu den folgenden drei Zeilen muss allerdings noch etwas erklärt werden:
Es besteht die Möglichkeit, den Spielbaum hart zu kodieren und eigene Klassen für diesen sowie seine Knoten zu erstellen. Die erste Idee dabei ist, jedem Knoten ein eigenes Spielbrett zuzuordnen, welches sich von dem seines Vaters an genau einer Position unterscheidet. Diese Position ist der aktuelle Zug. Diese Variante der Kodierung ist weit verbreitet. Aus mehreren Gründen wurde allerdings zur Implementierung des Baumes eine andere Herangehensweise verwendet. Recherchen[18] [19] bezüglich der Anwendung des Alpha-Beta-Algorithmus haben ergeben, dass es möglich ist, die Knoten eines Baumes mit einem temporären Spielbrett zu generieren, wodurch deutlich weniger Speicher benötigt wird. Diese Methode wird hier angewandt. Der Zug, welcher aktuell in der Schleife betrachtet wird, wird auf dem Spielbrett seines Vaters ausgeführt. Auf diesem Brett wird dann Negamax-Algorithmus mit negierten Parametern und einer decrementierten Suchtiefe aufgerufen, wobei der Rückgabewert im Stil des Negamax noch negiert wird. Dieser Aufruf generiert nun *rekursiv* alle Folgezüge bis zum Erreichen der Maximaltiefe und propagiert den bestmöglichen Wert zur Wurzel des Baumes zurück. Nach dem Aufruf des Algorithmus wird der gesetzte Zug einfach wieder vom Spielbrett entfernt und das selbe Brett kann für den nächsten Durchlauf der Schleife verwendet werden.
Die zweite Hälfte der Schleife von Zeile 15 bis Zeile 24 führt die **Alpha-Beta-Suche** durch. Dabei handelt es sich nicht, wie häufig in der Literatur [**?**, **?**, **?**] beschrieben um ein Verfahren, welches *nach* der kompletten Konstruktion des Baumes angewandt wird, um die Anzahl der Bewertungen zu verringern, sondern um eine Variante dieses Verfahrens, welche es ermöglicht *während* der Konstruktion schon überflüssige Teil-Bäume abzuschneiden. Es ist also eine weitere Beschränkung des Baumes. Ist der vom Folgezug hochpropagierte Wert *value* größer als der Wert des aktuell besten Knotens, so wird *value* zur neuen unteren Schranke für Zugwerte. Diese Anpassung des Suchfensters erhöht im weiteren Verlauf der Suche die Anzahl der Beschneidungen des Baumes. Beim Minimax-Algorithmus wird in der Max-Funktion die untere und in der Min-Funktion die obere Schranke angepasst. In der

[18] http://software-talk.org/blog/2012/01/muhle-das-brettspiel-inklusive-ki-mit-alpha-beta-algorithmus/

[19] http://de.wikipedia.org/wiki/Alpha-Beta-Suche

Negamax-Variante ist dies nicht sofort ersichtlich. Dennoch werden beide Schranken in der dazugehörigen Ebene angepasst, da beim Negamax-Aufruf die beiden Schranken negiert und vertauscht übergeben werden, wodurch der Wert der Variable *alpha* in jeder zweiten Ebene den Wert von *beta* annimmt. Anschließend wird der Folgezug mit seinem Wert *value* zum neunen Tupel *t*. Ist die neu gesetztes untere Schrank nun kleiner oder gleich der oberen Schranke, so wird ein Schnitt durchgeführt und der Teil-Baum wird nicht weiter betrachtet.

7.3.2 Heuristik des Reihenausgleichs

Da die Heuristik an das jeweilige Verfahren angepasst werden muss, wird für den Spielbaum eine andere Heuristik verwendet. Auch diese Heuristik bekommt ein Spielbrett übergeben und auch diese Heuristik bewertet mit Integer-Werten, doch stecken ein anderer Verwendungszweck und eine andere Herangehensweise dahinter. Während die in Kapitel 7.2.2 vorgestelle Heuristik einzelne Positionen auf einem Feld bewertet hat und aus diesen die beste ausgewählt hat, wird diese Heuristik dazu verwendet, den Zustand des gesammten Spielbretts zu bewerten. Da in einem kurzen Zeitraum viele verschiedene Spielbretter bewertet werden müssen, kann die Heuristik nicht so umfangreich wie die zuvor beschriebene gestaltet werden.
Die Idee hinter dieser Heuristik ist es, das Verhältnis zwischen eigenen und gegnerischen Reihen zu bewerten. Dabei wird zwischen verschiedenen Mustern von Reihen unterschieden. Diese Reihen werden in verschiedene Kategorienen bezüglich ihrer Gefähr für den Agenten oder seinen Gegenspieler eingeteilt. Jede der Kategorienen erhält bei der Bewertung ein eigenes Gewicht. Die Bewertungsfunktion selbst geht jede Position auf dem Feld durch und ruft die einfachen Reihen-Erkennungsfunktionen auf. Diese Funktionen zählen die jeweils erkannten Reihen. Am Ende der Bewertungsfunktionen wird die Anzahl der gefundenen Reihen einer Kategorie mit dem jeweiligen Gewicht multipliziert. Die daraus entstehenden Werte der eigenen Kategorienen werden addiert und die der gegnerischen subtrahiert. Aus dieser einfachen Rechnung ergibt sich das Verhältnis aller Reihen eines Feldes, welches dann zur Bewertung des Feldes dient.
Die Kategorien lassen sich allerdings noch in zwei Oberkategrien unterteilen. Zum einen gibt es Reihen, die nicht mehr blockierbar sind. Darunter fallen die Fünfer-Reihe und die Vierer-Reihe mit zwei unblockierten Enden. Zum anderen gibt es Reihen bei denen es noch möglich ist, sie in ein bis zwei Zügen abzuwenden. Reihen der ersten Kategorie erhalten beim Bewerten ein viel höheres Gewicht als die der zweiten. Die Kategorien und ihre Gewichte,

sowie die dazu gehörigen Arten von Reihen sind im Folgenden aufgelistet:

Kategorie	Gewicht	Art der Reihe
Kategorie 1	10	Einzeln Spielsteine
Kategorie 2	30	Zweier-Reihe
Kategorie 3	70	Dreier-Reihe mit Lücke
Kategorie 4	100	Unblockierte Dreier-Reihe
Kategorie 5	500	Einseitig blockierte Vierer-Reihe, Vierer-Reihe mit einerLücke
Kategorie 6	9980	Unblockierte Vierer-Reihe
Kategorie 7	10000	Fünfer-Reihe

Erklärung:
Kategorie 1 und 2 dienen ausschließlich zum Zählen des Verhältnisses der Spielsteine, falls noch keine weiteren Reihen existieren. In Kategorie 3 befinden sich Reihen, welche in den nächsten zwei Zügen zum Sieg führen können, jedoch durch einen einzigen Zug neutralisiert werden können. Reihen der Kategorie 4 ermöglichen den Sieg in zwei Zügen, können jedoch durch zwei gegnerische Züge neutralisiert werden. Befindet sich ein Zug in Kategorie 5 so kann der Spieler im nächsten eigenen Zug gewinnen. Der Gegner kann diese Reihen jedoch mit einem Zug neutralisieren. Ist es einem Spieler möglich eine Reihe der Kategorie 6 oder 7 zu generieren, hat er schon gewonnen. Daher ist das Gewicht dieser Reihen deutlich höher. Das Gewicht von Kategorie 6 ist jedoch immer noch minimal kleiner als das von Kategorie 7, da bei der Möglichkeit eine Fünfer-Reihe zu vervollständigen oder eine unblockierte Vierer-Reihe auszubauen, der direkte Sieg bevorzugt werden soll.

Listing 6: Auswertung der Reihen-Heuristik

```
1   int width = fld.getWidth();
2   int height = fld.getHeight();
3
4   int ownSingles = 0;
5   int oppSingles = 0;
6   int ownTwo = 0;
7   [...]
8
9   int evaluation = 0;
10
```

```
11
12    for(int i = 0; i < width; i++){
13      for(int j = 0; j < height; j++){
14
15        //calling every heuristic method
16        ownSingles = ownSingles + numberOfTokens(fld, i, j,
              getColor());
17        [...]
18        oppFive = oppFive + fiveRow(fld, i, j, getOpponentsColor()
              );
19        //gobang and pente
20        if(rules > 0){
21          ownBeat = ownBeat + beatTwo(fld, i, j, getColor());
22          oppBeat = oppBeat + beatTwo(fld, i, j, getOpponentsColor
                ());
23        }
24      }
25    }
26    //add the numbers of tokens multiplied by their weight
27    evaluation = Threat_One*ownFive - Threat_One*oppFive -
          Threat_Two*ownFour -
28    Threat_Two*oppFour + Threat_Three*ownBFour - Threat_Three*
          oppBFour +
29    Threat_Four*ownThree - Threat_Four*oppThree + Threat_Five*
          ownBrThree -
30    Threat_Five*oppBrThree + Threat_Six*ownTwo - Threat_Six*oppTwo
          +
31    Threat_Seven*ownSingles - Threat_Seven*oppSingles;
32
33    [...]
34
35    return evaluation;
36  }
```

In der geschachtelten For-Schleife werden die einzelnen Funktionen zum Erkennen und Zählen der Reihen aufgerufen. Im Code sind diese *fiveRow* und *numberOfTokens*. Danach werden alle gezählten Reihen einer Kategorie mit ihrem Gewicht multipliziert. *Threat_One* ist das Gewicht von Kategorie 1, *Threat_Two* das von Kategorie 2 usw.
Dieses System ermöglicht es, bei Bedarf noch weitere Reihenerkennungen den Kategorien hinzufügen zu können. Soll beispielsweise eine weiterr Reihe zur Kategorie 1 hizugefügt werden, so muss man nur die dafür erstellte Erkennungsfunktion F mit $ownSingles = ownSingles + F$ in der geschachtelten For-Schleife aufrufen. Es besteht also einfach die Möglichkeit, die *Informiertheit* der Heuristik zu steigern. Dabei muss allerdings darauf geachtet werden, dass die Heuristik nicht zu viel Zeit zur Bewertung eines einzelnen Spielfeldes benötigt.
Zum Abschluss der Erläuterung dieser Heuristik, wird nun noch einmal auf die von G. F. Luger [**?**] angeführten Maße zur Bewertung einer Heuristik eingegangen:
Da der Spielbaum so aufgebaut ist, dass nur immer genau ein Weg zu einem Knoten führt, sind sowohl die *Zulässigkeit* als auch die *Monotonie* im Sinne Lugers erfüllt. Allerdings ist der Spielbaum auch nicht wie der Lugers aufgebaut, wodurch die beiden Maße nicht vollständig sinngemäß anwendbar sind. Haben zwei Knoten die gleiche Bewertung und liegen in unterschiedlichen Ebenen, so könnte man allerdings eine von Luger vorgeschlagene Funktionen zur Bewertung der Weglänge verwenden und diese noch in die Bewertung mit einfließen lassen. Demnach hätte der Knoten in der höheren Ebene dann eine bessere Bewertung als der andere. Für diese Funktion wird abhängig von der Ebene des Knotens die Tiefe der Ebene als Faktor zur eigentlichen Bewertung des Knotens addiert.
Die *Informiertheit* der Heuristik ist im Verhältnis zu anderen Heuristiken mit 16 Funktionen zur Erkennung von Reihen noch realtiv gut. Auf Grund der Anzahl von zu bewertenden Spielfeldern werden aber nicht einzelne Positionen bewertet. Die Heuristik ist auf die Suche in Spielbäumen zugeschnitten. Würde man sie beispielsweise beim Influence Mapping verwenden, so wäre das Ergebnis nicht mehr nachvollziehbar.

7.3.3 Vor- und Nachteile

Ein Nachteil bei der Arbeit mit Spielbäumen ist, dass die Visualisierung eines solchen Baumes die CPU sehr stark beansprucht und in den meisten Fällen auf Grund der Größe des Baumes gar nicht möglich ist.

Auch der Zeitaufwand dieses Verfahrens ist verhältnismäßig hoch und wächst exponentiell mit der Anzahl der zu berechnenden Ebenen[**?**].
Ein Vorteil des hier verwendeten Verfahrens ist das große Optimierungspotential, das dem Alpha-Beta-Verfahren inne wohnt. Durch viele, sich vom Implementierungsaufwand stark unterscheidende, Ansätze kann die Suchtiefe noch gesteigert werden.
Der größte Vorteil der Umsetzung mit Spielbäumen ist die Möglichkeit, mehrere Züge vorauszusehen. Dies entspricht der menschlichen Spielweise, Züge im Voraus zu planen, welche in komplexen Spielen wie beispielsweise *Go* immer noch die effektivste Spielweise ist[**?**].

8 Praktische Tests

Die praktischen Tests dienen zum Vergleich der entwickelten Agenten untereinander und mit anderen Agenten. Es soll ein realistischer Eindruck davon gewonnen werden, wie effektiv die angewandten Verfahren und Heuristiken in der Praxis wirklich sind. In den folgenden Testszenarien wird der Agent, welcher Influence Mapping verwendet, als *Agent I* bezeichnet und der Agent, welcher Spielbäume verwendet als *Agent S*.
Die Tests werden mit einem Alienware Andromeda R5 durchgeführt, der über folgende Eigenschaften verfügt:

- Prozessor: Intel Core i5-2320 mit 3.00 GHz
- Arbeitsspeicher: 8.00 GB
- Betriebssystem: Windows 7 Home Premium 64-Bit, mit Service Pack 1

8.1 Online Vergleiche

Um die Spielstärke der Agenten mit unabhängigen Implementierungen vergleichen zu können, wurden Tests mit Agenten aus dem Web durchgeführt. Da die Entwickler dieser Agenten keinen oder nur sehr wenig Code zum Vergleich zur Verfügung stellen, können nur wenige Schlüsse auf den Unterschied der Spielstärke mit Hilfe von Code gezogen werden. Keiner dieser Agenten wurde bei der Entwicklung der in dieser Thesis vorgestellten Agenten als Vorbild genommen. Noch vor der Auswertung der Tests wird angemerkt, dass in keinem einzigen Testspiel das Ergebnis „Unentschieden“ erziehlt wurde.

Um dieses zu erreichen, müssten insgesamt mindestens 361 Züge[20] ausgeführt werden. Alle Spiele wurden aber in 30 bis 80 Zügen beendet.

8.1.1 „Pente“

Diese Implementierung[21] wird von ihren Entwicklern (David Kron, Matt Renzelmann, Eric Richmond und Todd Ritland) zwar Pente genannt, doch spielt sie nach den Regeln von Gomoku. Dennoch bietet sie viele Einstellungsmöglichkeiten und verschiedene Verfahren zur Zugauswahl. Außerdem stellen die Entwickler den Code der verwendeten Algorithmen auf ihrer Webseite[22] zur Verfügung. Vor dem Spielstart können sowohl die Größe des Spielbretts, die maximale Dauer eines Zuges, die Länge der zu konstruierenden Reihe als auch der Algorithmus ausgewählt werden, den der Agent verwenden soll. Zu Testzwecken werden folgende Einstellungen vorgenommen:

Parameter	Wert
Spielbrettgröße	19x19
Maximale Zugdauer	3 Sekunden
Verbinde-k	$k = 5$
Algorithmus	Alpha-Beta-Suche

Mit dieser Konfiguration wurden jeweil 1000 Testspiele gegen *Agent I* und *Agent S* durchgeführt.
Die Gewinnquote der beiden Agenten gegenüber dem Vergleichsagenten liegt jeweils bei 100%. *Agent S*, mit Suchtiefe 3, benötigt dabei zur Bestimmung eines Zuges durchschnittlich ˜226,3 Millisekunden[23], während *Agent I* durchscnittlich ˜5,9 Millisekunden[24] benötigt.
Der verhältnismäßig große Unterschied zwischen den beiden Zeiten ist durch die Anzahl der zu überprüfenden Felder zu erklären. Obwohl *Agent S* eine einfachere und performance-orientiertere Heuristik verwendet, muss diese immer noch ungefähr das 10.000-Fache an Spielfeldern überprüfen.
Die Spielergebnisse beider Agenten sind in diesem Test zufriedenstellend. Um die Agenten bezüglich ihrer Spielstärke noch besser einordnen zu können, werden die folgenden Test durchgeführt.

[20] Beim Gobang oder Pente durch die Schlagregel sogar mehr

[21] http://pages.cs.wisc.edu/˜mjr/Pente/ (22.8.2012)

[22] eben diese

[23] inklusive Erstellen und Durchsuchen des Spielbaums

[24] inklusive Aktualisierung der Influence Map

8.1.2 Auway-Gobang

Auway-Gobang ist eine Web-Anwedung[25], welche das Spielen gegen einem Agenten mit drei verschiedenen Schwierigkeitsgraden erlaubt. Auch bei Auway-Gobang handelt es sich wieder um eine Gomoku-Implementierung. Die Feldgröße ist auf 13x13 festgelegt. Zum einen soll bei diesen Tests geprüft werden, ob die umgesetzten Agenten auf einem kleineren Feld Probleme verursachen. Zum anderen ist bei diesem Agenten nicht erkennbar, welches Verfahren er zur Bestimmung der Züge verwendet, wodurch sich nur auf die Zeit zur Zugbestimmung und auf das Spielergebnis konzentriert werden kann.
Für die Tests wurde der Schwierigkeitsgrad „Leicht“ gewählt. Wieder wurden mit jedem Agenten 1000 Testspiele durchgeführt.
In den Testspielen gegen **Agent I** gewann der Agent von Auway-Gobang ungefähr 44% aller Spiele . Dabei fällt auf, dass *Agent I* häufig verliert, wenn er selbst den ersten Zug ausführt. Er gewinnt dafür meistens, wenn sein Gegenüber anfängt. Dies ist unerwartet, da in den verwendeten Brettspielen normalerweise der beginnende Spieler im Vorteil ist [**?**]. In ihrem Artikel [**?**] nennt Jenny Lam allerdings mehrere Erklärungen für das Phänomen.

- Zum einen besteht die Möglichkeit, dass auch der Agent von Auway-Gobang mit einer Baumstruktur spielt und Alpha-Beta-Beschneidung. Der Spieler spielt also nicht optimal. Es wird eventuell keine ausreichende Suchtiefe erreicht oder die Heuristik verwendet keine guten Gewichtungen.

- Zum anderen startet der zweite Spieler seine Suche von einem Spielbrett aus, welches über einen Stein mehr verfügt, als das des ersten Spielers. Daher ist der zweite Spieler immer in der Lage, bis zu einem tieferen Level im Baum zu suchen. Bei gleicher Suchzeit hat der zweite Spieler also immer einen Vorteil. Dieser Umstand wird *Horizon Effect* genannt[**?**].

Letzteres ist kein direkter Vorteil für die Influence Map, da sie nur eine Ebene in die Zukunft sieht, sondern nimmt nur dem Agenten von Auway-Gobang diesen Vorteil.
Beim Spiel auf dem kleinen 13x13-Brett benötigt *Agent I* durchschnittlich ˜1,17 Millisekunden[26] zur Wahl eines Zuges. Mit der genannten Gewinnquote von ca. 56% kann man als Entwickler zufrieden sein, da selbst die „einfache“ Variante von Auway-Gobang für menschliche Amateurspieler nicht leicht zu

[25] http://www.spielerei.net/spielen/auway-gobang/ (22.8.2012)

[26] inklusive Aktualisierung der Influence Map

schlagen ist[27]. Man würde erwarten, dass beide Spieler immer gleich spielen und daher die Gewinnquote beim 0% oder 100% liegen würde. Dies ist allerdings beim *Agent I* durch den verwendeten Zufallsfaktor beim Auftreten von mehreren Positionen mit Maximalbewertung nicht gegeben. Die Anwedbarkeit von *Agent I* auf kleineren Feldern ist damit bestätigt.

Die Testspiele gegen **Agent S** erziehlte *Agent S* eine Gewinnquote von ca. 50%. Dafür beötigt *Agent S* durchschnittlich ˜152,8 Millisekunden[28] mit einer Suchtiefe von 3. Bei einem einzelnen Testspiel mit Suchtiefe 5 benötigte *Agent S* druchschnittlich ˜2376,5 Millisekunden. Die 50%-Quote war davon abhängig, ob *Agent S* als erstes am Zug war. War dies der Fall, so verlor der Agent jedes Spiel. Als Begründung dafür wird wieder der im Artikel von Jenny Lam[**?**] angeführte *Horizon Effect* genannt, welcher es dem zweiten Spieler ermöglicht, eine Suchebene weiter zu sehen.

8.1.3 Mobiltelefon-Anwendung - Pente

Dieser Test wird mit einer Anwendung[29] für Android Mobiltelefone druchgeführt. Diese hat eine Gesammtgröße von 228 Kilobyte. Die Tests werden auf einem Huaway U8510 mit der Android-Version 2.3.3 durchgeführt.
Die Anzahl der Testfälle ist hier deutlich geringer, da sich die Tests mit dem Mobiltelefon als umständlich erweisen. Jeder Agent führt 100 Testpiele aus. Die Handyanwendung wurde gewählt, um den Vergleich mit einem Gegenspieler ziehen zu können, der von professionellen Spielern entwickelt wurde. Außerdem bietet die Anwedung die Möglichkeit, wirklich das Spiel **Pente** zu spielen, bei dem sowohl *Agent I* als auf *Agent S* auf eine erweiterte Heuristik zugreifen. Bei dem Spielfeld handelt es sich um ein 19x19-Brett und als Schwierigkeitsgrad wurde „einfach“ gewählt. Dabei handelt es sich um den geringsten Schwierigkeitsgrad, den die Anweundung zur Verfügung stellt.

- **Agent S**:
  Diesen Test beendet der Agent mit einer Gewinnquote von 0%. Diese lässt sich darauf zurückführen, dass sich die Heuristik des Reihenausgleichs nicht für Pente zu eignen scheint. Die Veränderungen des Spielbretts durch das Schlagen von Steinen erschweren die Berechnung von vorausberechneten Zügen. Obwohl *Agent S* in Testspielen gegen Amateurspieler im Pente eine vernünftige Leistung erzielt, ist er dem Agenten der Pente.org-Experten nicht gewachsen.

[27] wurde von 4 Amateuren mit keiner bis wenig Erfahrung im Gobang-spielen getestet

[28] inklusive Erstellen und Durchsuchen des Spielbaums

[29] pente.org v1.0.2 von Mark Mammel

- **Agent I**: Auch *Agent I* verlor 100% aller Spiele. Dies tritt hier häufig auf, da das Angreifen von gegnerischen Zweier-Reihen zu hoch bewertet wird und damit ein entscheidener Zug verloren geht.

Dieser Test hat gezeigt, dass beide Agenten im Spielmodus Pente noch über viel Verbesserungspotential verfügen. Allerdings ist dies aufgrund der einfachen verwendeten Verfahren nicht überraschend.

8.2 Direktvergleich der Agenten

Um die beiden entwickelten Agenten noch besser vergleichen zu können, werden sie noch einmal im direkten Spiel gegeneinander getestet. Dazu werden jeweils 500 Testspiele für Gomoku, Gobang und Pente durchgeführt, wobei sowohl *Agent S* als auf *Agent I* jeweils 250 Mal als erster am Zug sind.
Bei den Tests zeigt sich, dass *Agent S* mit Suchtiefe 3 durchschnittlich ˜2075,8 Millisekunden[30] zum Berechnen eines Zuges benötigt. Bei wenigen Spielsteinen liegt dieser Wert noch bei unter einer Sekunde, steigt aber mit zunehmender Anzahl an Spielsteinen an. *Agent I* hingegen benötigt durchschnittlich 6,4 Millisekunden[31]. Bezüglich der Geschwindigkeit ist der Agent, welcher Influence Mapping verwendet also mehr als 300 mal so schnell wie der Spielbäume verwendende Agent. Bei beiden wird das Spielfeld exakt gleich dargestellt. *Agent S* muss jedoch bei jeder Zugberechnung mehr als 10.000 mal so viele Spielfelder überprüfen wie *Agent I* und kann dies nur aufgrund der „abgespeckten“ Heuristik in der angegebenen Zeit bewerkstelligen.
Bezüglich der Spielstärke werden die drei Spiele unabhängig voneinander betrachtet:

- **Gomoku**:
  Agent I konnte ca. 52% der Spiele für sich entscheiden. Das Ergebnis ist also relativ ausgeglichen. Bei Betrachtung der Spiele fällt auf, dass sich diese eigentlich nur an einem bestimmten Zug unterscheiden. Setzt *Agent I* diesen zu seinem Vorteil, so gewinnt er, wählt er aber die andere Variante, so gewinnt *Agent S*. Das Ergebnis von 50% ist nachvollziehbar, da *Agent I* bei zwei Positionen, in der Influence Map, mit dem gleichen Wert mit einer Wahrscheinlichkeit von ca. 50% die erste der beiden Positionen wählt.

[30] inklusive Erstellen und Durchsuchen des Spielbaums

[31] inklusive Aktualisierung der Influence Map

- **Gobang**:
  In den Gobang-Partien trennten sich die beiden Agenten sogar mit einer jeweiligen Gewinnquote von ca. 50%. Die Begründung dafür ist die selbe, wie für das Ergebnis der Gomoku-Testspiele. Da das Schlagen beim Gobang nur indirekt[32] zum Siegt beiträgt, zieht *Agent I* keine Vorteile aus seiner informierteren Heuristik. Mit anderen Worten, spezielle Angriffszüge[33] haben keine großen Auswirkungen.

- **Pente**:
  Das Ergebnis der Pente-Partie unterscheidet sich deutlich von denen der anderen. Hier konnte *Agent S* fast 72% aller Spiele für sich entscheiden. Die agressive Spielweise des *Agenten I*, bedingt durch die hohe Bewertung des Schlagens von Spielsteinen, zahlt sich nicht aus. Dadurch verliert der Agent einige Züge, welche *Agent S* dazu nutzt, seine Fünfer-Reihe zu bauen. *Agent S* legt das Hauptaugenmerk weiterhin auf das Zählen von Reihen und Bauen von Reihen. Nur sehr wenige Spiele konnte *Agent I* durch das Schlagen von 5 Stein-Paaren des Gegners gewinnen. Daraus muss das Fazit gezogen werden, dass ein Spieler zwar durch das Schlagen Druck auf den anderen Spieler ausübt, diesen aber nur selten allein dadurch besiegen kann. Mit dieser Erkenntnis sollten die „Schlagregeln“ der Heuristik von *Agent I* geringer bewertet werden.
  50 Testdurchläufe mit den Änderungen der Bewertung der Funktionen zum Finden gegnerischer Zweier-Reihen[34] von 500 auf 400 und zum Schlagen dieser[35] von 600 auf 450 ergaben das Folgende:

  1. *Agent I* verlor 100% der Spiele
  2. Die Spiele dauerten im Durchschnitt 20 Züge länger

  Daraus lässt sich schließen, dass die Änderung eine defensivere, aber auch uneffektivere Spielweise zur Folge hat. Die Änderungen werden also wieder Rückgängig gemacht und es wird vermerkt, dass die Heuristik der Mustererkennung bezüglich ihrer Pente-Funktionen Verbesserungsbedarf hat.

Das Ergebnis der Testläufe ist überraschend. Während sich *Agent I* in den Testspielen gegen andere Agenten[36] gleich gut oder besser schlug als *Agent S*,

[32] Geschlagene Steine werden nicht gezählt

[33] z.B. Doppelangriff, Dread X

[34] findTwo(fld, getOpponentsColor(), i, j)

[35] possibleComplete(fld, getColor(), i, j)

[36] Testszenarien 1-3

spiegelt sich dies im direkten Vergleich nicht wieder. Im Pente verliert *Agent I* sogar deutlich. Alles in allem sind die Ergebnisse der Tests als zufriedenstellend zu bezeichnen.

9 Fazit

Zum Abschluss der Arbeit mit Agenten für die Brettspiele Gomoku, Pente und Gobang wird noch einmal erläutert, ob das Ziel der Arbeit erreicht wurde. Danach folgt ein Ausblick zu möglichen Verbesserungen der Agenten und abschließend nimmt der Autor noch einmal Stellung zur Bachelorarbeit.

Wurde das Arbeitsziel erreicht?

Wie am Ende der Einleitung erwähnt war das Ziel der Arbeit die Entwicklung von *nachvollziehbar* spielenden Agenten für die drei Brettspiele, wobei verschiedene Verfahren entwickelt und auf ihre Stärke und Performance getestet wurden.

Die Tests mit anderen Agenten haben gezeigt, dass die beiden im Rahmen dieser Arbeit entstandenen Agenten es von der Spielstärke her mit bisher exitierenden Agenten für die verschiedenen Spiele aufnehmen können. Die Tests dienen daher als Nachweis für die *Nachvollziehbarkeit* der Spielweise der Agenten.

Beide Agenten können fehlerfrei ausgeführt werden. Das beinhaltet das Spielen gegen die Agenten sowie das Spielen dieser untereinander. Das Arbeitsziel wird daher als erreicht betrachtet.

Mögliche Verbesserungen

Da beide Agenten noch keine hundertprozentige Gewinnquote gegen andere Agenten aufweisen und auch gegen menschliche Spieler häufig unterliegen, gibt es noch viel Raum für Verbesserungsmöglichkeit. Dieser wurde aufgrund der Abgabefrist der Arbeit nicht weiter genutzt. Allerdings wurden zu beiden Agenten Überlegungen angestellt, wie man die Spielstärke noch erhöhen könnte.

Agent mit Influence Mapping und Mustererkennung:

- Bezüglich der Heuristik lässt sich die *Informiertheit* noch deutlich steigern, indem **mehr Mustererkennungen** hinzugefügt werden. Da die Bestimmung eines Zuges sehr schnell geht, steht noch viel ungenutzte Rechenzeit zur Verfügung, bis die Zeit des anderen Agenten erreicht ist.

- Da das Influence Mapping auch über die Möglichkeit verfügt, die Wer-

te einzelner Positionen und benachbarte Positionen zu **propagieren**, kann dies genutzt werden, um an den Enden von Reihen, die mindestens Länge 2 haben, die Werte entsprechend dem Verlauf der Reihe weiterzugeben. Damit kann das Verfahren dann mehrere Züge im Voraus berechnen, da das Potential einer Reihe besser eingeschätzt wird.

Agent mit Spielbaum und Reihenausgleich:

- Das Alpha-Beta-Verfahren bietet viele Möglichkeiten zur Optimierung. Eine häufig verwendete ist die **Fenster-Suche**, bei der die Schranken Alpha und Beta bei der Initialisierung nicht minimal und maximal gewählt werden, sondern durch eine geschicktere Wahl der Schranken versucht wird, schon früh Schnitte im Baum zu erzeugen. Diese Optimierung birgt allerdings die Gefahr, dass bei zu enger Wahl der Schranken gute Teil-Bäume abgeschnitten werden. Aus diesem Grund wurde die sonst einfach zu implementierende Optimierung noch nicht angewandt.

- eine weitere Optimierung wäre das **Ordnen der Kinder** eines Knotens. Sind diese richtig geordnet, so treten noch mehr Schnitte auf und das Alpha-Beta-Verfahren kann sein volles Potential ausschöpfen.

- Würde man eine **Datenbank** mit bewerteten Anfangs- und Endzuständen des Spielbretts erstellen und diese in die Zugwahl mit einbeziehen, könnte man die Spielstärke des Agenten deutlich steigern. Dieses Verfahren verwendete auch *Deep Blue*, der erste Schachcomputer der 1996 den Weltmeister Garri Kasparov schlug[37].

- Es ist möglich den Agenten so zu verändern, dass er beim Spielen lernt. S. Russell und P. Norvig beschreiben in ihrem Werk *Artificial Intelligence: a modern approach*[**?**] einen Algorithmus der sich **Q-learning** nennt und sich zu diesem Zweck hervorragend eignet.

Alles in Allem erforderte die Entwicklung von zwei Agenten viel Zeit und ein hohes Maß an Recherche. Die meiste Zeit wurde dabei für die Planung und Implementierung der Heuristiken verwendet. Nach Abschluss der Arbeit werden nun noch einmal die Erkenntnisse zusammengefasst, welche Autor und Leser aus der Arbeit mitnehmen sollen.
Das Forschungsgebiet der „Künstlichen Intelligenz“ bietet viele verschiedene Verfahren und Lösungsansätze. Diese sind von grundverschiedener Natur und

[37]http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

haben unterschiedliche Vor- und Nachteile. Verwendet ein solches Verfahren Heuristiken, so ist die Leistung des Verfahrens zu einem Großteil von der verwendeten Heuristik abhängig. Schlussendlich zeigt sich auch, dass viel Expertenwissen nötig ist, um einen Agenten zu entwickeln, der komplexe Probleme konstant mit einem guten Ergebnis löst.