University of Applied Sciences Wedel

Arimaa

Studiengang: Medieninformatik

Bachelor-Thesis „Bachelor of Science“ (B.Sc.)

Bearbeitet von:	Maurice Tollmien – minf8914@fh-wedel.de
Eingereicht am:	25. Februar 2014
FH-Wedel Betreuer:	Prof. Dr. Gerd Beuster

Inhaltsverzeichnis

1 Einleitung

Seit Ende des Jahres 2002 taucht vermehrt ein neues strategisches Brettspiel für zwei Spieler namens Arimaa in den Medien auf [29] [51] [27]. Laut der Erfinder Omar Syed und Aamir Syed bietet das Spiel eine neue Möglichkeit, sich mit Intuition und strategischem Denken gegen den Computer durchzusetzen, nachdem andere Spiele wie Schach von Computern mittlerweile dominiert werden.

Diese Arbeit wird sich mit dem Thema Arimaa auseinandersetzen und dabei auf die möglichen Facetten eingehen, welche dieses Spiel aus der Masse der existierenden Brettspiele hervorhebt, wo seine Komplexität liegt und warum gerade Arimaa schwer für Computer und gleichzeitig einfach für Menschen sein kann.

Zur Verdeutlichung der Ausarbeitung und Demonstration der genannten Punkte wird parallel eine Engine entwickelt, in der die wichtigsten in der Arbeit auftauchenden Algorithmen implementiert und ausgiebig getestet werden. Das Ergebnis wird mit dem aktuellen Stand weiterer Bots und ähnlicher Arbeiten verglichen und die Unterschiede verdeutlicht.

1.1 Entwicklung und Vorgeschichte von Arimaa

Anfang Februar 1999 kam Omar Syed, ehemaliger Mitarbeiter des NASA Research Center, die Idee, ein neues Brettspiel zu entwickeln. Eine der Ursachen war die Niederlage von Garry Kasparov (ehemaliger Schach-Weltmeister und Großmeister) gegen den Schach-Super-Computer Deep Blue von IBM im bekannten Match im Jahre 1997. Durch den Umstand, dass Deep Blue ein ausschließlich für Schach entwickelter Super-Computer war, entstand der Eindruck Syeds, dass Garry Kasparov nicht durch die überragende Intelligenz seines Gegners verlor, sondern hauptsächlich wegen dessen extremer Rechenleistung [45].

Dadurch angespornt entstand die Idee, ein Spiel zu entwickeln, welches einerseits leichte Regeln besitzt, leicht zu lernen ist, sich intuitiv spielen lässt, doch andererseits sich nicht durch reine Rechenleistung eines Rechners bewältigen lässt [45]. Für die kommenden Jahrzehnte sollte Arimaa weiterhin ein von Menschen dominiertes Spiel werden, trotz der exponentiell wachsenden Rechenleistung aktueller Computer.

Hier setzt die Arimaa-Challenge (siehe 1.4 „Die Arimaa-Challenge“) an, um interessierte Programmierer und Entwickler von Systemen künstlicher Intelligenz dazu zu bewegen, sich an der Entwicklung von Bots für Arimaa zu beteiligen und damit in der Öffentlichkeit einen höheren Bekanntheitsgrad des Spiels zu erreichen. Nach diesen Vorgaben entstand, nach über einem Jahr an Entwicklung, das Spiel Arimaa, abgeleitet aus dem Namen Omar Syed's Sohn Aamir.

1.2 Spielaufbau und Regeln

Damit das Spiel vor allem für Kinder und Erwachsene einfach zu lernen und zu spielen ist, wurden die Regeln recht einfach gehalten. In den folgenden Unterkapiteln sind alle Regeln festgehalten und erklärt.

Das Spielfeld besteht aus 8×8 Feldern. Die Spalten des Bretts werden unterteilt in die Buchstaben *A* bis *H*, während die Reihen des Bretts als Zahlen dargestellt werden. Von unten nach oben mit den Zahlen *1* bis *8*. Das Feld unten links ist also das Feld *A1*, das Feld rechts daneben *B1*, während das Feld ganz oben rechts das Feld *H8* ist.

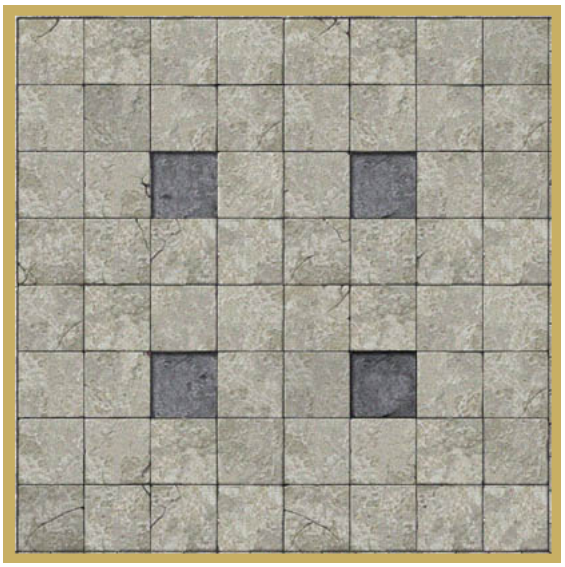


Abbildung 1: *Ein leeres Arimaa Spielfeld*

1.2.1 Traps

Es gibt vier Felder, die sich etwas von den anderen Feldern unterscheiden. Sie sind auf dem Spielfeld meist farblich hervorgehoben und befinden sich immer auf den Positionen *C3*, *F3*, *C6* und *F6*.

Diese Felder sind die einzigen Felder, auf denen Figuren gefangen genommen und aus dem Spiel entfernt werden dürfen. Weitere Informationen finden sich im Abschnitt 1.2.7 „Capture“.

1.2.2 Farben und Figuren

In dem Spiel Arimaa spielen zwei Spieler gegeneinander. Der Spieler, der anfängt, hat immer die Farbe Gold. Er setzt seine Figuren immer auf der südlichen Seite des Bretts (*A1* bis *H2*). Der andere Spieler hat die Farbe Silber und seine Grundstellung ist immer die nördliche, also *A7* bis *H8*.

Arimaa umfasst insgesamt zwölf verschiedene Figuren. Oder, wenn man die Farben nicht unterscheidet, sechs verschiedene Figur-Typen. Die verschiedenen Figuren haben eine unterschiedliche Rangordnung, welche bei dem eigentlichen Ziehen und Bewegen der Figuren von Bedeutung ist. Mehr dazu findet sich in den folgenden Kapiteln.

Im Spiel spielen beide Spieler mit insgesamt je 16 Figuren gegeneinander. Folgende Figuren kommen zum Einsatz:

 Gold Elefant	 Gold Kamel	 Gold Pferd
 Gold Hund	 Gold Katze	 Gold Hase
 Silver Elefant	 Silver Kamel	 Silver Pferd
 Silver Hund	 Silver Katze	 Silver Hase

Tabelle 1: *Alle Figurtypen in Arimaa*

Die Figuren kommen in einem Spiel pro Spielfarbe unterschiedlich oft vor. Der Elefant und das Kamel kommen je einmal vor. Das Pferd, der Hund und die Katze je zweimal und der Hase achtmal. Zwischen den einzelnen Figuren existiert eine folgende Rangordnung bezüglich ihrer Wertigkeit und Stärke:

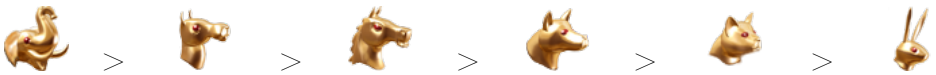


Tabelle 2: *Rangordnung der Figuren*

Für Silber erfolgt diese entsprechend.

1.2.3 Startaufstellung

Die Startaufstellung bei Arimaa unterscheidet sich stark von anderen Brettspielen ähnlicher Formate. Die Figuren können zu Beginn des Spiels beliebig auf die eigenen zwei Startreihen (für Gold Reihe 1 und 2, für Silber Reihe 7 und 8) gesetzt werden.

So kann eine angewendete Taktik im Spiel gleich zu Anfang durch eine entsprechende Aufstellung der Figuren unterstützt werden. Der goldene Spieler fängt immer an und setzt seine 16 Figuren. Danach darf der silberne Spieler seine Figuren setzen. Er kann in dem Fall auf die Aufstellung von Gold entsprechend reagieren und eine Aufstellung wählen, die unter Umständen Schwächen von Golds Aufstellung ausnutzt oder eine Strategie Silbers unterstützt.

Der Umstand, dass die Figuren beliebig gesetzt werden dürfen verhindert effektiv eine Vorberechnung eines Eröffnungsbuches, bei dem perfekte Eröffnungen gespielt werden, wie es bei Schach üblich ist.

Bei einem Elefant, einem Kamel, zwei Pferden, zwei Hunden, zwei Katzen und acht Hasen ergibt sich die folgende Anzahl möglicher unterschiedlicher

Startpositionen:

$$\#Startpositionen = \binom{16}{8} * \binom{8}{2} * \binom{6}{2} * \binom{4}{2} * \binom{2}{1} * \binom{1}{1} = 64.864.800$$

Die Formel resultiert daraus, dass wir bei 16 über 8 16 verschiedene Möglichkeiten haben, die 8 Hasen zu setzen. Bei den übrig gebliebenen 8 Positionen, können wir die 2 Katzen setzen. Es bleiben 6 freie Positionen übrig. Hier können die zwei Hunde beliebig gesetzt werden, auf die übrigen 4 Positionen die zwei Pferde. Für das Kamel bleiben noch 2 Möglichkeiten und für den Elefanten nur noch eine.

Damit sind alle Möglichkeiten abgedeckt, die zwei Startreihen zu belegen. Die Anzahl aller möglichen verschiedenen Startpositionen beläuft sich auf fast 65 Millionen. Da aber beide Spieler ihre Startpositionen beliebig setzen können, kann es insgesamt also

$$\begin{aligned} 64.864.800 * 64.864.800 &= 64.864.800^2 \\ &= 4.207.442.279.040.000 \\ &= 4,207 * 10^{15} \end{aligned}$$

verschiedene Startpositionen geben.

Es gibt also über 4 Billiarden Möglichkeiten für unterschiedliche Startpositionen. Vorberechnete Eröffnungsbibliotheken erübrigen sich also aufgrund der Vielzahl an Startmöglichkeiten und der folgenden Züge.

1.2.4 Schritt / Halbzug / Zug

Bei Arimaa und im Besonderen in der vorliegenden Ausarbeitung wird unterschieden zwischen einem Schritt, einem Halbzug und einem Zug. Der Schritt entspricht einer einzelnen Bewegung einer Figur um ein Feld und ist die kleinste Form einer Bewegung. Ein Halbzug beinhaltet vier Schritte einer Farbe und repräsentiert eine gesamte Bewegung einer Farbe in einer vorliegenden Situation. Ein Zug beinhaltet einen Halbzug der jeweiligen Spieler, also acht Schritte.

Die Schritte bei Arimaa sind sehr einfach. Alle Figuren haben die gleichen Bewegungsmöglichkeiten. Alle Figuren können sich in alle vier Richtungen bewegen. Also Norden, Süden, Westen, Osten beziehungsweise oben, unten, links, rechts.

Die einzige Ausnahme bilden die Hasen. Sie dürfen sich nicht zurück bewegen. Das bedeutet, dass sich ein goldener Hase nur nach oben, links und rechts bewegen kann, während ein silberer Hase nur nach unten, links und rechts ziehen darf. Alle Züge sind aus der Sicht des goldenen Spielers, mit dem Feld *A1* links unten.

Eine weitere Besonderheit von Arimaa ist, dass jeder Spieler in seiner Runde maximal vier Schritte mit beliebigen Figuren gehen darf. Ein goldenes Kamel auf dem Feld *D4* kann also, wenn Gold an der Reihe ist, mit vier Schritten folgende blau markierte Felder erreichen:

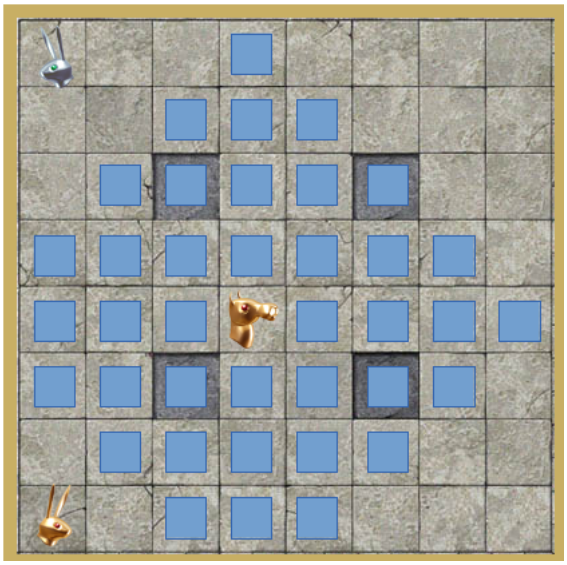


Abbildung 2: *Alle für das goldene Kamel erreichbaren Positionen in einem Halbzug*

Ein Hase hingegen kann in einem Halbzug nur folgende Positionen erreichen:

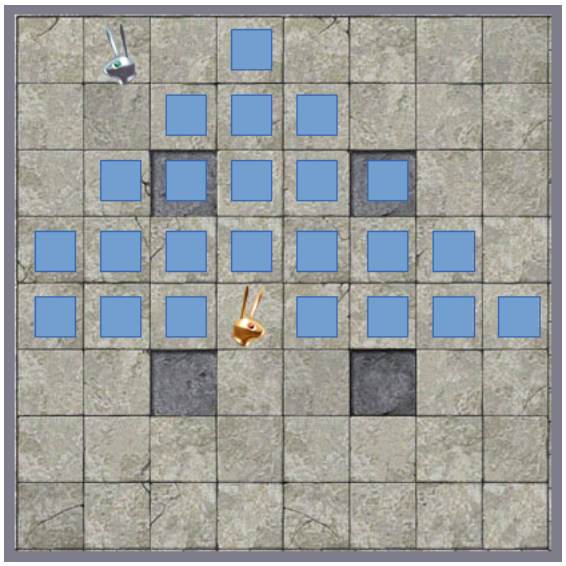


Abbildung 3: *Alle für den goldenen Hasen erreichbaren Positionen in einem Halbzug*

Für Silber sind die Züge entsprechend spiegelverkehrt.

1.2.5 Freezing

Das sogenannte Freezing bedeutet, dass eine Figur unter bestimmten Umständen nicht bewegt werden darf. Dies ist der Fall, wenn eine höherwertige gegnerische Figur neben der eigenen steht und gleichzeitig keines der drei angrenzenden Felder mit einer Figur mit der eigenen Farbe belegt ist.

Im Fall des Freezing darf die Figur während des eigenen Schritts nicht bewegt werden. Sie kann aber innerhalb des Halbzuges durch eine andere Figur der eigenen Farbe „befreit“ werden, indem diese neben die gefreezede Figur gezogen wird.

In der folgenden Grafik darf Silber das Pferd auf *D5* nicht bewegen, da der Elefant eine höherwertige gegnerische Figur darstellt und das silberne Pferd keine Unterstützung aus dem eigenen Team hat.

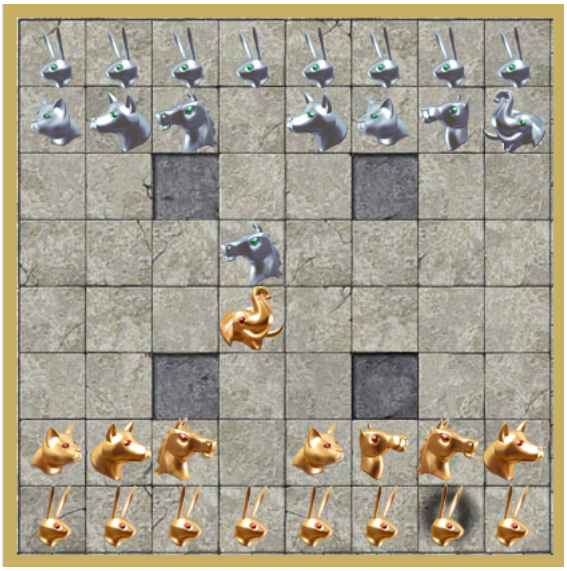


Abbildung 4: *Freezing des silbernen Pferdes*

Sobald aber eine weitere silberne Figur neben das silberne Pferd gezogen wird, darf dieses wieder bewegt werden.

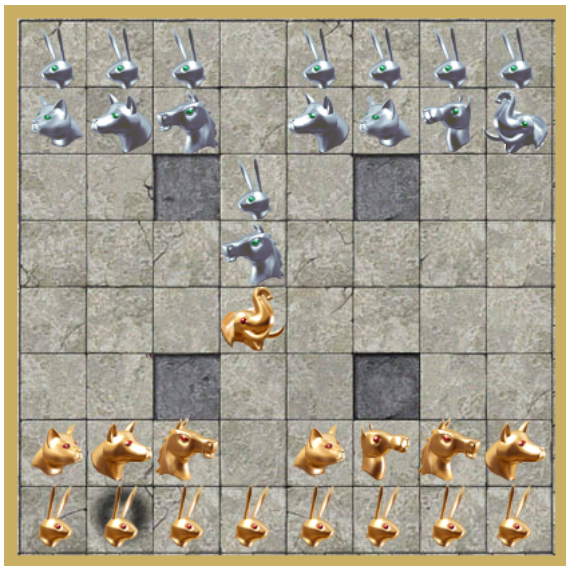


Abbildung 5: *Unfreezing des silbernen Pferdes*

1.2.6 Schieben / Ziehen

Das Schieben und Ziehen von gegnerischen Figuren ist eine Regel, in der gegnerische Figuren bewegt werden dürfen. Dies ist aber nur unter bestimmten Bedingungen erlaubt. Die eigene Figur, welche die gegnerische Figur ziehen oder schieben will, muss höherwertig sein als die gegnerische. Die eigene Figur darf nicht gefreezed sein (siehe Abschnitt 1.2.5 „Freezing“), und es muss entsprechend Platz zum Ziehen oder Schieben vorhanden sein.

Beim Schieben einer gegnerischen Figur bewegt sich die eigene Figur auf das Feld des Gegners und schiebt diese in eine der drei verbleibenden Richtungen um ein Feld. Die gegnerische Figur muss während dieses Vorgangs nicht gefreezed sein.

In folgender Grafik kann der goldene Elefant auf *D4* das silberne Pferd entweder auf *C5*, *E5* oder *D6* schieben. Der Elefant befindet sich danach immer auf der Position *D5*.

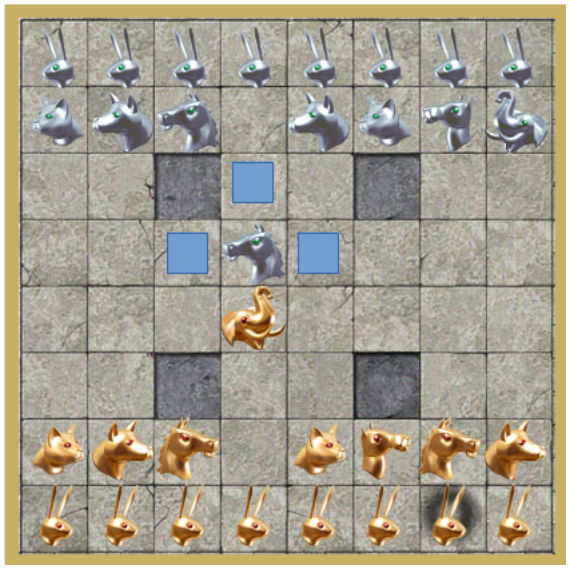


Abbildung 6: *Schiebe-Möglichkeiten für den goldenen Elefanten*

Im Gegensatz zum Schieben zieht man beim Ziehen eine gegnerische Figur auf die eigene Position. Die Bedingungen für das Ziehen sind identisch zum Schieben. Beim Ziehen aber wird die eigene Figur auf eines der drei übrigen möglichen Felder bewegt und die gegnerische Figur auf das eigene Feld gezogen.

In der folgenden Grafik kann der Elefant auf *D4* nach *C4*, *D3* oder *E4* bewegt werden. Das silberne Pferd wird dann immer auf *D4* gezogen.

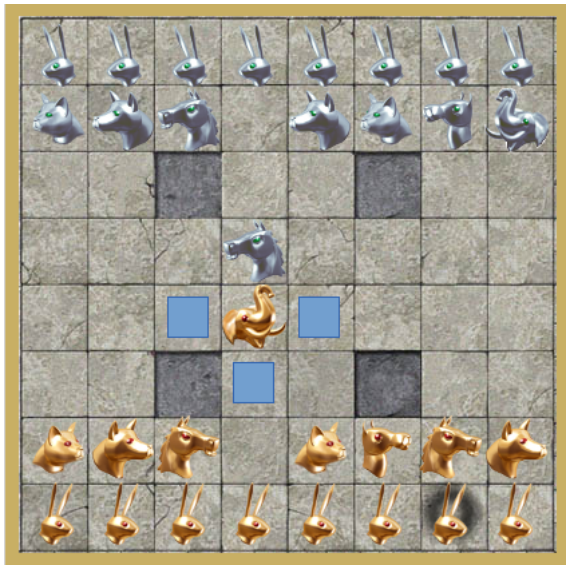


Abbildung 7: *Ziehe-Möglichkeiten für den goldenen Elefanten*

Eine letzte Besonderheit des Ziehens / Schiebens ist, dass der gesamte Vorgang als zwei einzelne Schritte gilt. Wenn der Spieler an der Reihe ist, können also maximal zwei ziehende/schiebende Züge getätigt werden.

1.2.7 Capture

Im Gegensatz zu Schach kann nicht während des Spiels an einer beliebigen Position eine gegnerische Figur geschlagen und vom Spielfeld entfernt werden. Bei Arimaa ist die einzige Möglichkeit, eine Figur gefangen zu nehmen (vom Spielfeld zu entfernen), wenn diese auf einem Trap steht (siehe Abschnitt 1.2.1 „Traps“) und nicht durch eine Figur aus dem eigenen Team unterstützt wird, das bedeutet, wenn keine Figur der gleichen Farbe neben dem Trap steht. Erst dann wird eine Figur entfernt.

Im Gegensatz zu anderen Spielen ist dadurch auch das Entfernen eigener Figuren, ohne jegliche Einwirkung des Gegners, möglich. Im folgenden Beispiel ist das goldene Pferd, welches auf dem Trap auf *C3* steht, durch den goldenen Hund auf *B3* gesichert. Sobald sich dieser aber in jeglicher Form bewegen würde, müsste das goldene Pferd vom Spielfeld entfernt werden, es sei denn eine weitere goldene Figur würde sich vorher neben den Trap platzieren.

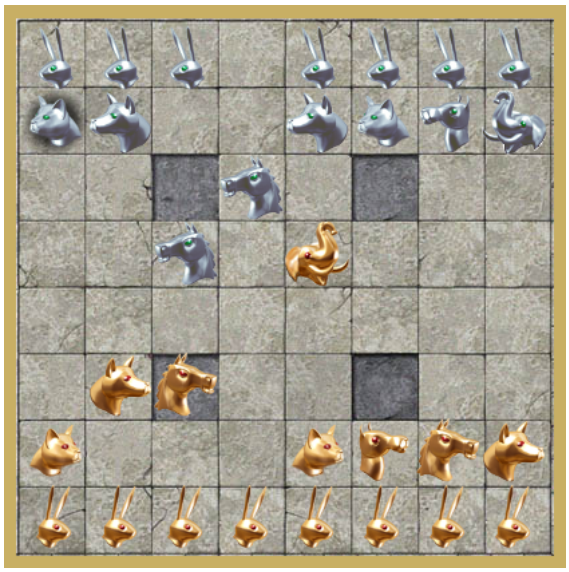


Abbildung 8: *Das goldene Pferd auf einem gesicherten Trap*

Da die Traps die einzige Möglichkeit darstellen, Figuren aus dem Spiel zu entfernen, stellen die Traps eine Schlüsselposition im Spiel dar. Wenn man in der Lage ist, diese zu kontrollieren, kann man leicht fremde Figuren in diese ziehen/schieben und damit gefangen nehmen oder schlagen.

1.2.8 Gewinnsituationen

In Arimaa gibt es verschiedene Wege das Spiel zu gewinnen. Das eigentliche Ziel des Spieles jedoch ist es, einen eigenen Hasen auf die Grundlinie des Gegners zu bewegen.

Für Gold bedeutet das, er hat gewonnen, sobald sich am Ende eines Halbzuges ein goldener Hase auf der achten Reihe befindet. Für Silber muss am Ende eines Halbzuges ein silberner Hase auf die erste Reihe kommen. Hierbei spielt es keine Rolle, ob dieser dann gefreezed (siehe Abschnitt 1.2.5 „Freezing“)

ist. Es spielt weiterhin keine Rolle, wie der Hase dorthin kommt. Auch wenn er von dem Gegner dorthin geschoben/gezogen wurde.

Eine weitere Möglichkeit, den Gegner zu besiegen, ist, wenn dieser keine Möglichkeit hat, eine Figur zu bewegen. Das bedeutet zum Beispiel, wenn alle Figuren gefreezed oder durch andere Figuren blockiert sind. Wenn ein Spieler in seinem Halbzug keine einzige Figur bewegen kann, hat er verloren.

Ein Spieler verliert außerdem in dem Moment, wo er keine Hasen mehr zur Verfügung hat; das bedeutet, wenn alle acht Hasen gefangen genommen wurden. In dem Moment ist das eigentliche Ziel, einen Hasen auf die Grundlinie des Gegners zu bewegen, nicht mehr möglich, und der Spieler hat verloren.

1.3 Arimaa im Vergleich zu Schach

Oft wird beim Thema Arimaa Schach im gleichen Atemzug genannt. Zumindest entsteht vielfach der Eindruck, dass beide Spiele sehr verwandt seien. Dies ist wohl hauptsächlich darauf zurückzuführen, dass die Entwicklung Arimaas ähnliche Grundsätze wie Schach verfolgt, beides auf der gleichen Art Brett gespielt werden kann und Arimaa sogar mit Schachfiguren spielbar ist, wenn man die Figuren umbenennt.

Der Grundsatz Arimaas ist es, ein Spiel zu haben, welches für einen Computer schwer zu berechnen ist und gleichzeitig von Menschen relativ intuitiv spielbar ist. Schach hatte diesen Ruf lange inne. Als jedoch im Jahre 1997 das erste Mal ein Computer gegen einen Großmeister im Schach gewann (Garry Kasparov vs. Deep Blue (IBM)), schien der Bann gebrochen.

Wie im Abschnitt 1.3.1 „Schach-Programme ebenbürtig zu Großmeistern“ nachzulesen ist, sind Computer und Schachprogramme mittlerweile so weit, mittels künstlicher Intelligenz und weniger Rechenleistung, einem Schachgroßmeister Parole zu bieten.

1.3.1 Schach-Programme ebenbürtig zu Großmeistern

In den letzten 10 bis 15 Jahren hat sich nicht nur die Hardware exponentiell entwickelt. Auch Schachprogramme wurden ununterbrochen weiterentwickelt, neu entworfen und unaufhörlich verbessert. Seit der von IBM entwickelte Super-Computer Deep Blue, bei dem sogar die Hardware auf das Berechnen von Schachzügen ausgelegt und optimiert war, den damaligen Weltmeister und Schach-Großmeister Garry Kasparov schlug, hat sich viel getan. Deep Blue war in der Lage, mit seinen 32 CPU's eine gewaltige Anzahl von mehr als 200.000.000 Positionen pro Sekunde zu evaluieren [33] . Die so erreichte Rekursionstiefe war wohl der Hauptgrund für den Erfolg und den Gewinn gegen Kasparov 1997.

Deep Blue gehörte zu seiner Zeit 1997 sogar zu einer der besten und stärksten Super-Computer weltweit [17]. Trotzdem war der Computer Kasparov nicht haushoch überlegen, und die gespielten Spiele waren hart umkämpft. Gerade der Fakt, dass Kasparov vor dem Match ein Einblick in gespielte Partien von Deep Blue verweigert wurde [28], führte zu einiger Diskussion. Denn selbstverständlich hatte Deep Blue bei seiner Entwicklung Einblick in von Kasparov gespielte Partien, welche analysiert und ausgewertet wurden.

Zusätzlich dazu wurde Deep Blue sogar während des Matches gegen Kasparov von Entwicklern auf das Spielverhalten Kasparovs angepasst [33]. So entstand der Eindruck, dass es sich um ein unter unfairen Umständen ausgetragendes Match handelte, zugunsten von Deep Blue.

Die Entwicklung bei Schachprogrammen ging jedoch weiter. Unabhängig von der stärker werdenden Hardware wurden auch die implementierten Strategien besser und leistungsstärker.

Im Jahre 2003 trat Garry Kasparov, der jetzt 18 Jahre Nummer #1 weltweit im Schachspiel war, gegen das Schachprogramm „Deep Junior“ an. Es entstand ein Match, welches in einem unentschiedenen 3-3 endete [41]. Das wichtigste an diesem Match war jedoch nicht das Unentschieden des Schachprogrammes Deep Junior gegen den Schach-Weltmeister Kasparov, sondern die Umstände des Turniers. Besonders war, dass Deep Junior nicht, wie Deep Blue 1997, auf für das Schachspiel ausgelegte Hardware und einem Super-Computer ausgeführt wurde, sondern auf einem relativ konventionellem Computer, welcher über nur 8 Intel 1,6 GHz CPU und 8GB RAM verfügt [10]. Damit wurde gezeigt, dass ein auf entsprechende Intelligenz ausgelegtes Schach-Programm auch ohne dedizierte Hardware in der Lage sein kann, einen Top-Menschen in einem Turnier zu schlagen oder gleichauf zu sein.

Ein weiterer Erfolg des Computers gegen den Menschen in einem Schachturnier fiel im Jahre 2006 zu Gunsten des Schachprogramms “Deep Fritz“ gegen den Schach-Großmeister Wladimir Kramnik. Schon in vorherigen Matches gegen Deep Fritz konnte Kramnik 2002 nur ein Unentschieden erreichen. Bei dem eigentlichen Turnier musste sich der Schach-Großmeister 2-4 gegen den Bot Deep Fritz geschlagen geben [40]. Im Verhältnis zu anderen Turnieren lief das Schachprogramm Deep Fritz während des kompletten Turniers auf einem Dual Intel Core 2 Duo 5160, welcher sogar im Verhältnis zu heutigen Laptops keinem Vergleich stand hält [40]. Das bemerkenswerteste des Turniers war allerdings der Umstand, dass Kramnik ein strategischer Vorteil gewährt wurde.

Wladimir Kramnik bekam mehrere Monate vor dem eigentlichen Turnier den Quelltext des Bots Deep Fritz zur Einsicht, um ihn zusammen mit erfahrenden Programmierern auf mögliche strategische Schwächen untersuchen zu können. Die eigentlichen Unbekannten des Turniers waren ausschließlich die endgültige Leistung des Bots und ein modifiziertes Eröffnungsbuch [40].

Doch auch mit diesem doch relativ guten Vorsprung Kramniks gelang es ihm nicht, auch nur ein Spiel gegen Deep Fritz zu gewinnen. Vier Spiele liefen auf ein Unentschieden hinaus, während er die anderen beiden verlor.

Damit, so scheint es, ist eine Aera zu Ende gegangen, in der Menschen das Schachspiel dominieren. Denn sogar auf Standard-Rechnern sind gute Schachprogramme offensichtlich in der Lage, Schach-Großmeister in einem Turnier zu besiegen.

Ein weiteres und sehr aussagekräftiges Beispiel der Überlegenheit des Computers und der Schachprogramme zeigt uns der Bot „Pocket Fritz 4“. Im Jahre 2009 gelang es der Schach-Engine Pocket Fritz 4, das Großmeister-Level zu erreichen, mit einem ELO-Rating von 2898 [34]. Der Engine gelang es, ein

Turnier der Kategorie 6 (Copa-Mercosur-Turnier in Buenos Aires, Argentinien) mit neun gewonnenen und einer unentschiedenen Partie zu gewinnen [34]. Dabei lief der Bot die gesamte Zeit über auf einem handelsüblichen HTC Touch HD mit einem 528 MHz Prozessor. Seine Performance entspricht in etwa der des Deep Blue von IBM, er wertet aber nur 20.000 mögliche Positionen pro Sekunde aus [34].

Spätestens mit dieser Performance auf der Hardware ist endgültig aufgezeigt, dass Computer den Menschen in Schach mittlerweile endgültig überholt haben. Und das nicht nur durch eine Brute-Force Herangehensweise durch Auswertung immens vieler Positionen, sondern auch strategisch.

1.3.2 Der Einfluss von Bobby Fischer auf die Schach-Geschichte

Wie schon im Abschnitt 1.3.1 „Schach-Programme ebenbürtig zu Großmeistern“ erwähnt wurde, bietet Schach keine unüberwindbare Herausforderung mehr für Computer gegen Menschen. Arimaa ist also der nächste Schritt, ein Spiel zu gestalten, welches für Computer schwierig zu berechnen, jedoch intuitiv zu spielen sei für Menschen.

Doch Omar Syed und Aamir Syed waren nicht die Ersten mit diesem Gedanken. Schon im Jahre 1996 gab es Überlegungen in ähnliche Richtungen. So erfand Bobby Fischer, Schach-Großmeister, Legende und bezeichnet als einer der größten Schachspieler, der jemals gelebt hat, eine Variante von Schach namens „Chess960“.

Die Besonderheit von Chess960 ist, dass die Startaufstellung nicht komplett festgelegt ist. So sind die Startpositionen auf der jeweiligen ersten/letzten Reihe zufällig. Ursprünglich wurde diese Variante von Schach auch *Shuffle Chess* genannt. Bei der endgültigen Variante Chess960 existieren noch weitere Restriktionen bezüglich der Startaufstellung, welche im Endeffekt genau 960 verschiedene Startaufstellungen ermöglichen. Im Jahre 2008 wurde diese Variante von der FIDE als Zusatz in die Regeln des Schach aufgenommen [7].

Ein besonderer Vorteil dieser Variante des Schachs ist, dass ein Auswendiglernen von Eröffnungszügen nicht mehr in dem Maße möglich ist, wie es bei normalem Schach der Fall ist. Der amerikanische Wissenschaftler prägte in dem Zusammenhang den Begriff *Chunking Theory* [4]. Dabei geht es um die Untersuchung, wie viele Informationen, in unserem Fall Schachpositionen, sich ein Mensch merken und einprägen kann. Nach einigen Untersuchungen ist bekannt, dass sich Schach-Großmeister zwischen 50.000 und 100.000 verschiedene Positionen und Aufstellungen im Langzeitgedächnis einprägen konnten [4].

Anders ausgedrückt wird man Schwierigkeiten haben, ein Großmeister im Schach zu werden, wenn man nicht in der Lage ist, sich immens viele Eröffnungsspiele und andere Spielzüge einzuprägen.

Damit ist Schach aber nicht mehr nur ein Intelligenzsport, sondern beruht stark auf der Fähigkeit des Auswendiglernens. Mit der Einführung zufälliger Startpositionen wie im Chess960 oder komplett beliebigen Startaufstellungen wie in Arimaa macht man ein Ausweniglernen von Eröffnungszügen nahezu unmöglich. Stattdessen muss ab dem ersten Zug strategisch gespielt werden. Der

Spieler ist dann weniger davon abhängig, eine perfekte Eröffnung zu spielen, um im Mittelspiel noch eine Chance zu haben.

Ein weiterer Vorteil von zufälligen Startaufstellungen ist auch, dass bei Schachprogrammen keine festen Eröffungsdatenbanken angelegt werden können, um ein perfektes Anfangsspiel zu erlauben. Gerade gegen einen Computer muss im Schach ein perfektes Eröffnungsspiel erfolgen, um sich nicht schon am Anfang eine Blöße zu geben. Denn spätestens der Computer nutzt jede Gelegenheit und Schwäche des Gegners radikal aus. Und sei es mit hardcodierten Eröffnungsbibliotheken.

1.4 Die Arimaa-Challenge

Wie schon im Abschnitt 1.1 „Entwicklung und Vorgeschichte von Arimaa“ erwähnt wurde, ist einer der Intentionen hinter der Entwicklung des Spiels Arimaa, die Forschung im Bereich der künstlichen Intelligenz voran zu treiben und unter Umständen Techniken zu entwickeln, die auch in anderen Gebieten der künstlichen Intelligenz eingesetzt werden können.

Aus dieser Bestrebung heraus entstand auch die so genannte *Arimaa Challenge* [44] [46]:

> ***The challenge:*** *The first person, company or organization that develops a program which can defeat the top human Arimaa players in an official Arimaa challenge match before the year 2020 will win the Arimaa challenge prize. The prize is currently $11,000 USD. The official challenge match will be between the current best program and three selected human players.* (Omar Syed)

Die Challenge hat mehrere Hintergründe. Natürlich soll mit Hilfe des Geldwertes hinter einem Gewinn der Challenge der Bekanntheitsgrad des Spiels Arimaa erhöht werden. So sollen mehr Entwickler und Programmierer sich mit Themen der künstlichen Intelligenz auseinandersetzen und damit eine Möglichkeit bekommen, nicht nur die Arimaa Challenge zu gewinnen, sondern auch den Bereich der künstlichen Intelligenz voranzutreiben. So können mit Hilfe dieses Spiels unter Umständen neue Technologien entwickelt oder die Erforschung dieser voran getrieben werden.

Besonders Teilbereiche der Informatik, welche sich mit beispielsweise Neuronalen Netzen, Expertensystemen, Mustererkennung und evolutionären Algorithmen befassen, wurden im Bezug auf Arimaa wenig beachtet und könnten einen Einsatzbereich bei der Entwicklung von Bots in Arimaa, gleichzeitig aber auch in vielen anderen Bereichen der künstlichen Intelligenz und Informatik darstellen.

Dabei geht es vor allem um eine Weiterentwicklung durch Software und nicht performanter werdende Hardware. Um das zu gewährleisten, wird der Bot bei der Arimaa Challenge auf einem Linux-System ausgeführt, dessen Standard-Hardware maximal 1000$ kostet [44].

Sollte ein Bot die Challenge gewinnen, so wird der Preis nur dann ausgezahlt, wenn das Programm und seine Lösungswege in einem wissenschaftlichen Artikel beschrieben, dokumentiert und der ICGA bereitgestellt werden.

Die drei menschlichen Verteidiger der Challenge werden von Omar und Aamir Syed persönlich aus den besten Spielern ausgewählt. Zusätzlich werden zwei Backup-Spieler ausgewählt, die einspringen können, sofern einer der Hauptverteidiger der Challenge nicht in der Lage sein sollte, anzutreten.

Die Namen der Spieler werden erst nach der eigentlichen Challenge veröffentlicht, um eine Spezifizierung der Bots auf die Spielweise eines bestimmten menschlichen Spielers zu verhindern.

1.5 Vergleichbare Spiele

Generell lassen sich mit Arimaa viele Spiele vergleichen, welche zu der Familie der abstrakten strategischen Spiele gehören. Anders ausgedrückt alle Brett- oder Kartenspiele mit perfekter Information, bei der kein Zufall oder körperliches Talent gefordert ist. Meist werden solche Spieler von zwei Partien gegeneinander gespielt.

Ein paar bekanntere, deterministisch berechenbare Spiele mit ähnlicher Komplexität und vollständiger Information sind:

- Schach
- Shogi
- Weitere Schach Varianten
- Go
- Arimaa
- ...

Spiele wie *Mao*, *Calvinball* und *Seven Minutes in Heaven* gehören zwar nicht zu der gleichen Kategorie wie Arimaa, Schach und Go, gehören aber auch zu den Spielen, in denen Computer große Schwierigkeiten haben, gute Ergebnisse zu erzielen. Das liegt aber hauptsächlich an der Natur der Spiele mit varianten Regeln und Aktionen.

1.6 Kritikpunkte an Arimaa

Neben der vielen positiven Eigenschaften von Arimaa gibt es selbstverständlich kritische Meinungen gegenüber dem Spiel. Die wichtigsten und zentralen Kritikpunkte werden in diesem Abschnitt erläutert und diskutiert.

Viele der Diskussionspunkte beziehen sich auf den Fakt, dass es bisher keinem Computer gelungen ist, die Arimaa Challenge (siehe Abschnitt 1.4 „Die Arimaa-Challenge“) zu gewinnen oder in einem World Championship die besten Menschen zu schlagen [52].

Dabei wird vielfach der Vergleich zu Schach gezogen, oft bezogen auf den Fakt, dass bei Schach Computer regelmäßig in der Lage sind, Menschen in Wettkämpfen zu besiegen.

Der größte Kritikpunkt gegenüber Arimaa ist, dass Arimaa ein verhältnismäßig junges Spiel ist. Demnach gibt es entsprechend nur eine kleine Anzahl

an Programmierern, die sich der Herausforderung stellen und Bots für Arimaa entwickeln. Außerdem fallen jegliche Sponsoren und Geldsummen kleiner aus als in Schach-Wettbewerben. Schach hingegen existiert sehr viel länger. Tausende Entwickler haben sich seinen Herausforderungen gestellt und über 40 Jahre Schachprogramme weiterentwickelt. Demnach sei ein Vergleich zwischen Arimaa und Schach nicht gerechtfertigt. Die Behauptung ist, dass Arimaa genauso zu schlagen wäre, wenn die Bedingungen denen von Schach entsprächen.

Aber genau weil Arimaa ein junges Spiel ist, ist die Community Arimaa spielender Menschen verhältnismäßig sehr klein. Und auch die Menschen, die als Weltmeister und best-bewertete Spieler darstehen, spielen Arimaa meist erst ein paar Jahre und sind, im Verhältnis zu Schach-Großmeistern, noch immer als Laien zu bewerten. Gerade durch die im Verhältnis unerfahrenen Arimaa-Spieler haben Bots die Möglichkeit, mit deutlich weniger Aufwand und Fachwissen, die Arimaa-Challenge zu gewinnen.

Zusätzlich ist zu bemerken, dass die Programmierung von Arimaa-Bots der von Schach-Bots sehr nahe ist. Sowohl die Suchalgorithmen (siehe 3 „Grundalgorithmen“) als auch die Datenstrukturen (siehe 2.2 „Datenstruktur“) und Performance optimierenden Algorithmen wie Transposition Tables (siehe 4.3 „Transposition Table“) können ohne jegliche Änderung in ein Arimaa-Programm übernommen werden. Hier entfällt also jeglicher Forschungsaufwand, wie er bei Schachprogrammen seinerzeit nötig war, um performante Bots zu entwickeln.

Ein weiterer Punkt, der dem Kritikpunkt widerspricht, ist, dass bereits zu sehr früher Zeit in der Geschichte Arimaas sehr starke und erfahrene Entwickler angefangen haben, Bots zu entwickeln. Ein Beispiel ist der Entwickler David Fotland. Er entwickelte den Bot „Bomb“ , welcher in den Jahren zwischen 2004 und 2008 jede Computer-Weltmeisterschaft gewann [47]. In der eigentlichen Arimaa-Challenge blieben die Ergebnisse jedoch eindeutig [47].

Jahr	Herausforderer	Verteidiger	Ergebnis	Gesamt
2004	Bomb / David Fotland	Omar Syed	0-8	0-8
2005	Bomb / David Fotland	Frank Heinemann	1-7	1-7
2006	Bomb / David Fotland	Karl Juhnke Greg Magne Paul Mertens	0-3 0-3 1-2	1-8
2007	Bomb / David Fotland	Karl Juhnke Omar Syed Brendan M N Siddiqui	0-3 0-3 0-2 1-0	1-8
2008	Bomb / David Fotland	Jean Daligault Greg Magne Mark Mistretta Omar Syed	0-3 0-3 0-1 0-2	0-9

Tabelle 3: *Arimaa Challenge 2004-2008*

David Fotland gilt als eine Prominenz mit viel Erfahrung in der Entwicklung von Bots und künstlicher Intelligenz, da er der Entwickler von „Many Faces of Go“ ist. Ein Bot für das Spiel Go, welcher viele Jahre in Folge die Weltmeisterschaft gewann und als einer der stärksten Go-Bots überhaupt gilt [50].

Wie leicht zu erkennen ist, gelang es dem besten Bot, entwickelt von dem Profi David Fotland, nicht, die Challenge zu gewinnen. Im Gesamtergebnis erreichte sein Bot eine Maximalanzahl von einem gewonnenen Spiel von acht bis neun gespielten.

Mittlerweile gibt es stärkere Bots als Bomb. Jedoch zeigen die Ergebnisse und das Engagement von sehr erfahrenen Entwicklern, dass schon sehr früh in der Geschichte Arimaas ein hohes Niveau erreicht und performante Bots entwickelt wurden. Die letzten Arimaa Challenges der Jahre 2012 und 2013 zeigen, dass auch heute noch der Mensch eine deutliche Überlegenheit in Arimaa aufweist [47].

Jahr	Herausforderer	Verteidiger	Ergebnis	Gesamt
2012	Briareus / Ricardo Barreira	Jean Daligault „hanzack“ Eric Momsen	0-3 0-3 3-0	3-6
2013	Marwin / Mattias Hultgren	Mathew Brown Greg Magne Matthew Craven	0-3 0-3 1-2	1-8

Tabelle 4: *Arimaa Challenge 2012-2013*

Das Ergebnis aus 2012 war das beste je erreichte Ergebnis eines Bots in der Arimaa-Challenge [47]. Aber noch im Jahre 2013 wurde die Überlegenheit der Menschen dem Bot Marwin gegenüber mit einem 1:8 für die Menschen aufgezeigt.

Der nächste Kritikpunkt an der Arimaa-Challenge ist, dass im Schach bei einem Zusammentreffen zwischen Mensch und Maschine der Computer meist nur die Hälfte der Spiele gegen den Menschen gewinnen muss, während bei Arimaa 2/3 aller Punkte gegen die menschlichen Verteidiger erreicht werden müssen [46].

Das traf auf wenige Ausnahmefälle zu, bei denen ein Großmeister gegen ein spezifisches Schachprogramm angetreten ist. In normalen Turnieren finden jedoch normale Spiele mit Vorauswahl und mehreren Spielen statt, wie man es gewohnt ist [34]. Eine Ausnahme sind Spiele, wie die Begegnung von Deep Blue und Garry Kasparov [37] [33].

Ein weiterer Punkt aus der Liste der Kritiken gegenüber der Arimaa Challenge ist, dass der herausfordernde Computer vor der Challenge eine Reihe von Qualifizierungsspielen durchlaufen muss. Dies ermöglicht den Menschen, also auch den menschlichen Verteidigern der Challenge, sich mit der Spielweise der Bots vertraut zu machen, um mögliche Schwächen des Bots zu finden, aufzudecken und auszunutzen.

Das ist jedoch leicht zu widerlegen, da der Umstand, den gegnerischen Bot zu untersuchen, auch bei dem Schachspiel zwischen Kramnik vs Deep Fritz [40] (siehe 1.3.1 „Schach-Programme ebenbürtig zu Großmeistern“) im Jahre 2006 möglich war. Doch auch nach monatelanger Recherche von Deep Fritz, bei dem auch Einsicht in den Quellcode gewährt wurde, konnte der Großmeister Kramnik Deep Fritz nicht besiegen.

Auch der Einwand, dass bei Arimaa die Bots während des Matches nicht von Entwicklern angepasst werden durfte, bezieht sich schlussendlich nur auf das Match zwischen Deep Blue und Kasparov [37] [33]. Im Laufe späterer Matches zwischen Computerprogrammen und Großmeistern wurde während der Spiele selbstverständlich kein Code mehr verändert oder angefasst. Details finden sich im Abschnitt 1.3.1 „Schach-Programme ebenbürtig zu Großmeistern“.

Die Auswahl der Hardware, welche bei der Arimaa-Challenge eingesetzt wird, bildet einen weiteren Ansatzpunkt für Kritik. Für die Weltmeisterschaften und die Arimaa-Challenge werden alle Bots gleichermaßen auf einem Standardcomputer ausgeführt, welcher für rund 1000$ für jedermann zu erwerben ist [46] [44]. Im Gegensatz dazu ist bei Schachturnieren teilweise ein Einsatz ungewöhnlich starker oder schwacher Hardware erlaubt. Ein Beispiel ist das Match zwischen Kasparov und Deep Blue [37] [33].

Mittlerweile entkräftet sich dieses Argument von selber, da die letzten großen Schachspiele zwischen Computern und Großmeistern alle auf Standard-Hardware ausgeführt wurden. So auch im Beispiel von Kramnik vs Deep Fritz [40] (siehe 1.3.1 „Schach-Programme ebenbürtig zu Großmeistern“) oder Kramnik vs Deep Junior [10]. Eine noch bessere Demonstration der Stärke und Überlegenheit zeigt aber, dass auch Schachprogramme, welche auf extrem schwacher Hardware laufen, durch intelligente Programmierung, noch immer Großmeister-Level erreichen können [34] (siehe 1.3.1 „Schach-Programme ebenbürtig zu Großmeistern“).

1.7 Taktische und Strategische Herausforderungen

Wie auch in Schach und vielen weiteren Brettspielen kann bei Arimaa zwischen taktischen und stragetischen Herausforderungen unterschieden werden. Eine taktische Herausforderung bezieht sich auf eine kurzfristige und konkrete Verbesserung der eigenen Situation auf dem Spielfeld. Meist sind Taktiken nicht weiter als zwei Züge im Vorraus geplant. Ein Beispiel einer Taktik ist das Gefangennehmen einer gegnerischen Figur in seinem Halbzug. Das taktische Ziel der Gefangennahme ist damit erreicht.

Eine strategische Herausforderung befasst sich mit einer vorausschauenden, längerfristigen Verbesserung der Spielfeldsituation. Strategien sind umfassen meist deutlich mehr als zwei Züge. Ein Beispiel einer Strategie ist eine Übernahme und eines gegnerischen Traps ohne die Aufgabe eines Eigenen. Diese Form einer Strategie umfasst meist zehn oder mehr Züge und umfasst keine konkrete Abfolge von Schritten.

Eine Taktik ist in den meisten Fällen von Computern perfekt zu berechnen und auszuführen. Trotz des hohen Branching-Faktors sind in einer gegebenen Situation ein bis zwei Züge fast immer in der gegebenen Zeit berechenbar. Taktisch ist der Computer dem Menschen also gleich auf oder überlegen.

Durch den extrem hohen Branching-Faktor, die Komplexität des Spiels und die Anzahl möglicher Strategien ist eine Strategie meist nicht vollständig berechenbar. Das bedeutet, dass es für Computer schwer ist, auf langfristig angelegte Strategien menschlicher Spieler zu reagieren, da diese meist weit außerhalb des Berechnungshorizontes liegen.

Ein Beispiel einer Strategie ist eine langfristige Blockierung eines Elefanten durch das Opfern einer Katze. Der Computer erkennt, dass eine gegnerische Katze ungeschützt neben dem Trap steht. Taktisch ist die Gefangennahme der Katze sinnvoll. Strategisch gesehen kann der Elefant nach der Gefangennahme jedoch zu beiden Seiten blockiert werden. Er kann sich daher nur noch weiter hinter die gegnerischen Linien bewegen oder dort stehen bleiben. In beiden Fällen kann in dem folgenden Halbzug des Gegners der Elefant vollständig

blockiert werden.

Langfristig hat also der menschliche Spieler die höchste freie Figur auf dem Spielfeld. Die Erkennung der Strategie fällt einem Computer schwer, da diese drei bis fünf Züge beinhaltet und somit weit außerhalb des Berechnungshorizontes liegt.

Die strategischen Aspekte des Spiels ermöglichen Menschen, den Computer weiterhin zu besiegen und zu dominieren.

2 Konstruktion eines Arimaa-Bots

Bei der Konstruktion eines Arimaa-Bots ist nicht nur der eigentliche Algorithmus relevant, sondern insbesondere das Design des gesamten Projektes.

2.1 Klassenkonstrukte

Die Wahl zwischen verschiedenen Klassenstrukturen, Vererbungen und Instanzierungen bilden die Grundlage eines Arimaa-Bots. Sie ist relevant, um eine gute Abstraktion zu schaffen, die es ermöglicht, Algorithmen ohne Aufwand auszutauschen oder sogar während der Ausführungszeit dynamisch zu laden oder zu wechseln. Denn gerade bei komplexen Programmen ist es wichtig, einen guten Überblick zu behalten, um Fehler leicht zu finden oder Programmstellen leicht erweitern zu können.

Bei der Konstruktion und dem Design des Bots sind folgende Punkte zu berücksichtigen:

- Sehr einfache Main-Funktion, welche grundsätzliche Instanzen festlegt und die Kommunikation mit dem Server startet.
- Eine einfache Server-Kommunikation über StdIn, die ununterbrochen in der Lage sein muss, Nachrichten zu empfangen und zu verarbeiten.
- Eine Engine, die unabhängig von jeglicher Server-Kommunikation den besten Zug unter gegebenen Umständen errechnet.
- Ein einfacher Controller, welcher die Engine kontrolliert. Dazu gehören die Hauptfunktionalitäten wie starten, beenden und das Abfragen von Lösungen.
- Ein weiterer Controller, der die Engine ausschließlich abhängig von der abgelaufenen Zeit selbstständig beenden und Lösungen ausgeben lassen kann.
- Ein Modul zur Ausgabe von Nachrichten nach Stdout, welches unter Umständen gleichzeitig ein Logging beinhaltet. Dieses muss von allen Klassen und Threads gleichzeitig bedient werden können.
- Ein Modul mit globalen Attributen zum Spiel. Dieses muss von den meisten Klassen gleichzeitig zugegriffen werden können und wird von der Server-Kommunikations-Klasse konfiguriert.

In dem folgenden UML-Diagram ist eine Klassenstruktur und das grundsätzliche Design abgebildet, welches alle geforderten Punkte erfüllt und eine abstrakte Kapselung der einzelnen Komponenten, besonders der Algorithmen, berücksichtigt.

Abbildung 9: *UML-Diagram zur Abbildung der gewählten Klassenstruktur*

Die Wahl und Konzeption der Klassenstruktur beruht auf eigener Überlegung und beruht in keiner Form auf bestehenden wissenschaftlichen Ausarbeitungen oder Entwicklungen bestehender Arimaa Bots.

2.2 Datenstruktur

Der nächste wichtige Punkt, vor dem Entwurf eines Such-Algorithmus', ist die Wahl einer Datenstruktur. Im Fall von Arimaa ist dabei folgendes zu berücksichtigen:

- Das Feld besteht aus $8 \times 8 = 64$ Feldern.
- Zwei Parteien spielen mit je 6 verschiedenen Figurtypen gegeneinander.
- Figuren haben je nach Typ unterschiedliche Bewegungsregeln.

Zu berücksichtigen bei der Wahl einer Datenstruktur und Schnittstelle zu dieser ist auch, dass auf diese bei der Berechnung des bestmöglichen Zuges fast ununterbrochen zugegriffen wird und einen wesentlichen Teil der Laufzeit ausmacht.

Hier lohnt es sich also, eine effiziente/optimale Datenstruktur zu ermitteln, um im Wesentlichen eine gute Performanz zu erreichen, die ab diesem Moment nur noch maßgeblich von der Implementierung des Algorithmus' und des Algorithmus' selber abhängig ist und nicht von der zu Grunde liegenden Datenstruktur.

Weiterhin muss bei dem zu traversierenden Baum eine große Menge an Knoten untersucht werden zur Ermittelung des bestmöglichen Ergebnisses. Es ist also nicht nur die Performanz, sondern auch die Speichernutzung ein Kriterium zu Auswahl der Datenstruktur. Bezüglich der Datenstruktur bestehen große Ähnlichkeiten zum Schachspiel. Die grundlegende Problematik bei der Wahl einer Datenstruktur ist die gleiche (Performance, Speichereffizienz) [9].

Eine weit verbreitete Datenstruktur, die beiden Kriterien weitestgehend gerecht wird, und welche sowohl in Implementierungen von Arimaa, als auch Schach bereits verwendet wird, ist die sogenannte 'Bitboard'-Datenstruktur. Hierbei wird jedes einzelne Feld des Spielfeldes auf ein einziges Bit eines 64-bit großen Datentyps (Beispiel in Java: long) abgebildet. Und zwar jeder einzelnen Figur. Zusammen ergibt das eine Definition der gesamten Spielfeldsituation durch insgesamt 12 long Variablen (sechs Figuren a zwei Farben). Der Vorteil des genutzten Speichers erschließt sich sofort. Für jedes Feld wird nur ein Bit an Speicher benötigt.

Aber auch die Performance ist sehr hoch, da nahezu alle Überprüfungen einzelner Felder, möglicher Bewegungsrichtungen von Figuren und Aufstellungen von Figuren zueinander auf die Grundoperationen (shift, logisches Oder, logisches Und, logisches Not, logisches Xor) auf Bitebene zurückgeführt werden können. Die genannten logischen Bit-Operationen können effizient von der Hardware ausgeführt werden (parallelisiert). Sie sind damit ungleich schneller, als eine auf Klassen und Methoden basierende Datenstruktur.

Die Bitboard-Datenstruktur aus Schachprogrammen lässt sich für Arimaa zwar nicht komplett übernehmen, jedoch die Idee. Bezüglich der einzelnen Zugriffe und Methoden wurde die Idee der Bit-bezogenen Datenstruktur auf Arimaa angepasst und neu implementiert.

2.3 Zeitkontrolle

Die Zeitkontrolle ist ein sehr wichtiger Aspekt der Implementierung einer Arimaa-Engine. Grundsätzlich geht es darum, die Engine nach einer festgelegten Zeitspanne abbrechen zu können, um ein valides Ergebnis zurückgeben zu können. Außerdem kann es sein, dass die Engine manuell vom Controller durch den 'stop'-Befehl beendet wird (normalerweise nach abgelaufener Zeit, in Ausnahmefällen früher). Es gibt mehrere grundsätzliche Möglichkeiten, solch eine Zeitkontrolle zu realisieren. Alle vorgestellten Formen einer Zeitkontrolle sind eigenständig erarbeitet und beruhen nicht auf bereits veröffentlichen Arbeiten.

2.3.1 Primitiv

Die relativ einfach und primitive Möglichkeit wäre es, während der rekursiven Berechnung des besten Zuges (Traversieren des Baums mit Spielzuständen) immer aktuell den momentan besten Halbzug global abzuspeichern. Damit wäre dieser global zugreifbar von der Zeitkontrolle, und diese könnte den Thread der Engine durch ein einfaches 'interrupt();' beenden und den Halbzug auslesen. Hier ist allerdings zu berücksichtigen, dass der Interrupt-Befehl während einer Schreibe-Operation auf den Best-Move auftreten kann und diesen undefiniert (halbfertig) lässt und so einen invaliden Halbzug erzeugen kann. Zusätzlich ist es möglich, die Engine direkt nach dem Start abzubrechen, wenn noch kein Halbzug vorhanden ist.

Da der Thread aber interrupted wurde, gibt es keine einfache Möglichkeit für einen sogenannten 'Panic-Search' mehr (Berechnung eines Zuges mit sehr flacher Rekursionstiefe, der extrem schnell berechnet werden kann und einen validen Halbzug erzeugt). Zu den genannten Nachteilen und Problemen ist dieser Ansatz eine aus Software-Design-Sicht eine extrem unschöne und unsaubere Lösung.

2.3.2 Dynamisch

Eine weitere Möglichkeit wäre eine dynamische Anpassung der Zeitkontrolle beim Traversieren des Baumes. Das heißt, dass bei einer gewählten Rekursionstiefe zwischen 1/4 und 1/16 des Auswertungsbaumes ausgewertet wird. Danach checkt man, wie die genutzte Zeit zu der gesamt-möglichen Zeit für einen Halbzug verhält und passt die Rekursionstiefe für die weiteren Teilbäume so an, dass entweder mehr Zeit oder weniger Zeit genutzt werden kann. Hierbei kann es allerdings dazu kommen, dass unterschiedliche Teilbäume unterschiedlich tief ausgewertet werden. Da jedoch die Rekursionstiefe nicht konstant ist, ist nie sichergestellt, dass der gewählte Halbzug auch tief genug ausgewertet wurde.

Der größte Nachteil ist eher, dass die Funktion zur Traversierung des Baumes nun auch noch zusätzlichen Ballast durch eine zu verändernde Zeitkontrolle bekommt, welche die Funktion schnell unübersichtlich und groß machen wird. Der zusätzliche Berechnung-Overhead sollte nicht entscheidend ausfallen. Ein Vorteil hingegen ist, dass der komplette Lösungsbaum relativ konstant tief ausgewertet werden würde. Man würde keine Möglichkeit komplett übersehen können (Wie in vorheriger Methode beschrieben).

2.3.3 Statisch

Eine dritte und einfachste Möglichkeit wäre eine Anpassung der Rekursionstiefe vor der eigentlichen rekursiven Auswertung des Baumes. Dies könnte, abhängig von der konstanten Zug-Zeit, der Anzahl der Figuren auf dem Feld und anderer Faktoren geschehen, die den Branching-Factor des Spiels beeinflussen (siehe Abschnitt 5 „Branching-Faktor“). Durch vorberechnete Worst-Case Szenarien könnte sich relativ sicherstellen lassen, dass der gesamte Auswertungsbaum ausgewertet werden könnte; zum Beispiel könnte festgelegt werden, dass bei einer Zug-Zeit von 30 Sekunden und 32 Figuren auf dem Feld eine Rekursionstiefe von 6 angewendet wird, während bei nur noch 16 Figuren eine Rekursionstiefe von 8 angebracht ist.

Der große Nachteil dieser Variante ist, dass in der eigentlichen Traversierungsfunktion ein Overhead vorhanden ist, welcher die Funktion unübersichtlich oder zu komplex macht. Trotzdem ist relativ sichergestellt, dass alle Züge berechnet würden. Ein möglicher Nachteil ist jedoch, dass unter Umständen nicht die volle Zeit ausgenutzt werden wird, wenn man auf Nummer sicher geht und die Rekursionstiefe zu gering gewählt hat. Oder im anderen Falle, bei zu hoch gewählter Rekursionstiefe, ein kompletter Teilbaum nicht mehr ausgewertet werden kann, wenn die Zeit abgelaufen ist. Der bestmögliche Halbzug würde hier von der Funktion selber zurückgegeben werden und sich beim rekursivem Aufstieg zusammensetzen.

Die Abbruchkriterien wären dann entweder die Rekursionstiefe oder die abgelaufene Zeit. Im letzteren Fall kann es sein, dass ein Teilbaum mit einer Rekursionstiefe von 1 oder 2 bewertet werden würde. Wenn dieser dann das beste Ergebnis hätte, würde nur ein einziger Schritt als Bestmove zurückgegeben werden.

3 Grundalgorithmen

Dieser Abschnitt behandelt einige bekannte und oft verwendete Algorithmen, die allesamt ausschließlich zur Traversierung eines Zustandsbaumes dienen. Der Zustandsbaum stellt eine Teilmenge eines gerichteten Graphen in einem Suchraum zu dar. Durch die Spezifizierung als Teilmenge eines Suchraumes ist davon auszugehen, dass der vollständige Zustandsbaum durch seine Größe und Komplexität nicht in einer gegebenen Zeitspanne traversierbar ist.

So beschränken sich alle folgenden Suchalgorithmen auf entweder eine gegebene Zeitspanne, eine feste maximale Rekursionstiefe oder gegebenenfalls beides. Die vorgestellten Algorithmen dienen in diesem Fall der Traversierung eines Suchbaumes in einen Zwei-Spieler-Null-Summen-Brettspiel mit vollstän-

diger perfekter Information. Das Ziel jeder Suche ist es also, einen Knoten zu finden, der dem Spieler ein maximale Entlohnung bietet. Sie sind jedoch auch in der Lage, durch leichte Umstrukturierung, normale gerichtete oder ungerichtete Graphen zu traversieren.

Alle folgenden Algorithmen können in drei Kategorien aufgeteilt werden:

- ***Depth-First-Search*** *Bei der Traversierung des Teilbaumes werden Kindknoten immer vor Geschwister-Knoten berücksichtigt. Es entsteht eine asymmentrische Suche, welche immer bis zu der maximalen Rekursionstiefe voranschreitet und sich meist rekursiv von links nach rechts arbeitet.*

- ***Breadth-First-Search*** *Bei der Traversierung des Teilbaumes werden Geschwisterknoten immer vor Kindknoten berücksichtigt, was dazu führt, dass ein Baum immer symmetrisch von der Wurzel abwärts traversiert wird.*

- ***Best-First-Search*** *Diese Form der Baum-Traversierung gehört zu den Breath-First-Search-Algorithmen, bei der jedoch alle Knoten einer Rekursionstiefe abhängig ihres Wertes sortiert traversiert werden, so dass im Idealfall der beste Pfad (Folge von Knoten, welche zum maximierten Endknoten führt) immer im jeweils ersten Teilbaum liegt.*

3.1 Minimax

Der Minimax-Algorithmus ist grundsätzlich eine Methode zur Traversierung von Suchbäumen mit der Absicht, einen idealen Zweig (Principal Variation) mit der besten Evaluierung des Endknotens zu berechnen. Im Bereich Spieltheorie der theoretischen Informatik geht es hierbei hauptsächlich um eine Evaluierung von Spielzuständen und den dazugehörigen möglichen Zügen der Spieler.

Der Minimax-Algorithmus ist ausschließlich im Bereich von zwei Spielern und 'zero-sum' Spielen anzuwenden. Das bedeutet, dass für Spieler S1 für eine gegebene Strategie ein bester Halbzug V existiert und gleichzeitig den besten Halbzug für Spieler S2 bei der gleichen Strategie $-V$ darstellt [26].

Davon ausgehend minimiert der Minimax-Algorithmus den maximalen Verlust eines Spielers beziehungsweise maximiert den minimalen möglichen Ertrag, um somit ein gutes Gesamtergebnis zu erhalten. Der Minimax-Algorithmus bildet die Grundlage vieler weiterer Methoden der theoretischen Informatik im Bereich der Spieltheorie.

Eine Brett-Konstellation eines Arimaa-Spiels kann durch einen Baum schematisiert werden. Die jeweiligen Teilbäume bilden sich aus allen möglichen Situationen, die sich durch einen Schritt auf dem Spielfeld ergeben.

Ein Beispiel eines beschriebenen Baumes mit einer Rekursionstiefe $d = 3$ und einem Branching-Faktor $b = 2$ wäre folgender:

Die Zahlen in den jeweiligen Endknoten repräsentieren das Ergebnis einer Evaluation eines Boardzustandes. Der rot gekennzeichnete Teilbaum mit dem evaluierten Wert 7 zeigt hierbei die sogenannte Principal-Variation an (Weitere Details im Abschnitt 6 „Principal Variation“).

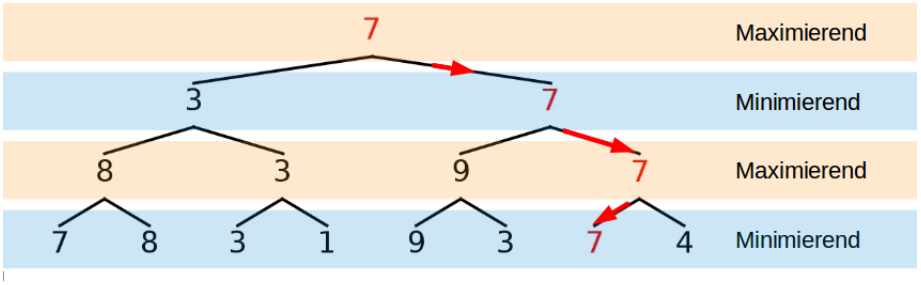


Abbildung 10: *Einfacher Suchbaum für Arimaa*

Definition 1 *Ein Zweig des Spiels A^j ist definiert als ein Tupel (S, succ), für das gilt:*

- ***S*** *Die Menge aller von der gegebenen Situation ausgehend möglichen Spielkonstellationen.*
- ***succ*** *: S —> Reg(S) Funktion, die alle Teilbäume (Kinder) s_j einer Brett-Konstellation s_i ermittelt, wenn ein gültiger Schritt exisitiert, der das Brett von s_i auf s_j verändert.*

Definition 2 *Die Suchbäume sind die Teilbäume einer Spielkonstellation festgelegt durch eine gegebene Tiefe d.*

Definition 3 *Ein Teilbaum des Suchbaumes ist festgelegt durch seinen Grad b und seine Tiefe d, wenn gilt, dass:*

- *Alle nicht-terminal Knoten haben durchschnittlich b Kinder.*
- *Alle Terminalknoten befinden sich an der Tiefe d.*

Definition 4 *Ein Knoten gilt als nicht-terminal, wenn er mindestens einen Kindknoten hat.*
Ein Knoten gilt als terminal, wenn er keinen Kindknoten besitzt.

Definition 5 *Eine heuristische Evaluationsfunktion $f : S \to Z \mid \{Z \in \mathbb{R}$, S ist eine mögliche Spielfeldsituation$\}$, f gibt also den inhärenten Wert einer Spielkonstellation zurück.*

Der Minimax-Algorithmus versucht in jeder Rekursionstiefe für den jeweiligen Spieler das Ergebnis zu maximieren, ausgehend davon, dass das Maximum für Spieler 1 dem Minimum von Spieler 2 entspricht. Der Minimax-Algorithmus arbeitet rekursiv aufsteigend und damit rückwärts vom Endknoten zur Wurzel des Suchbaumes. In jedem Schritt des maximierenden Spielers wird geprüft, ob das Ergebnis des evaluierten, traversierten Teilbaums besser als das bisherige beste Ergebnis ist. In dem Fall wird der ehemals beste Wert überschrieben. Für den Gegner (entspricht dem minimierenden Spieler) wird das Ergebnis

minimiert und ist ansonsten entsprechend.
Dabei wird der gesamte Suchbaum traversiert und alle Endknoten evaluiert. Der folgende Pseudo-Code trägt zum allgemeinen Verständnis bei:

```
minimax (TreeNode node, int depth, boolean maximizingPlayer)

    if depth == 0
        return evaluierten Wert des aktuellen Board-Zustandes

    moveList = generateAllMoves()
    if maximizingPlayer
        for (Move move : moveList)
            node = applyMove()
            score = max (score, minimax (node, depth - 1, !maximizingPlayer))
    else
        for (Move move : moveList)
            node = applyMove()
            score = min (score, minimax (node, depth - 1, !maximizingPlayer))

    return score
```

Abbildung 11: *Pseudo-Code für Minimax*

Da der komplette Baum über die gegebene Tiefe d traversiert werden muss, um ein valides Ergebnis zu erzielen, haben wir, ausgehend von einem konstanten Branching-Faktor b (siehe Abschnitt 5 „Branching-Faktor“), b^d Knoten zu traversieren. Das entspricht einer exponentiellen Zeit-Komplexität von $O(b^d)$. Genauere Tests bezüglich der verschiedene Algorithmen im Vergleich finden sich in Kapitel 8 „Tests“.

3.2 Alpha Beta Pruning

Der Algorithmus Alpha Beta Pruning ist eine Erweiterung des oben beschriebenen Minimax-Algorithmus' (siehe: 3.1 „Minimax“). Die grundlegende Methodik der Traversierung des Suchbaumes bleibt gleich. Eine maßgebliche Verbesserung bildet jedoch der Bestandteil des sogenannten Prunings. Hierbei können Teilbäume des Suchbaumes ignoriert werden, wenn sichergestellt ist, dass, egal, was dabei herauskommt, das Ergebnis keinen Einfluss auf die weitere Berechnung hat.

Ein Beispiel wäre folgender Baum:

Wie in der Abbildung zu erkennen ist, können die gelb markierten Knoten, egal ob sie ausgewertet werden oder nicht, keinen Einfluss auf das Gesamtergebnis haben. Dieser Effekt kommt dadurch zustande, dass ein Ergebnis, je nach Spieler, entweder minimiert oder maximiert wird. In unserem Beispiel versucht also der minimierende Spieler (Gegner), sein Ergebnis zu minimieren, während der maximierende Spieler seine Ergebnisse maximiert. Wenn aber ein Zwischenergebnis des minimierenden Spielers bereits kleiner als ein Zwischenergebnis des darüber liegenden maximierenden Spielers ist, spielt es für diesen keine Rolle mehr, ob der minimierende Spieler ein für sich besseres Ergebnis (minimierendes) findet oder nicht, da bei der darauffolgenden Maximierung

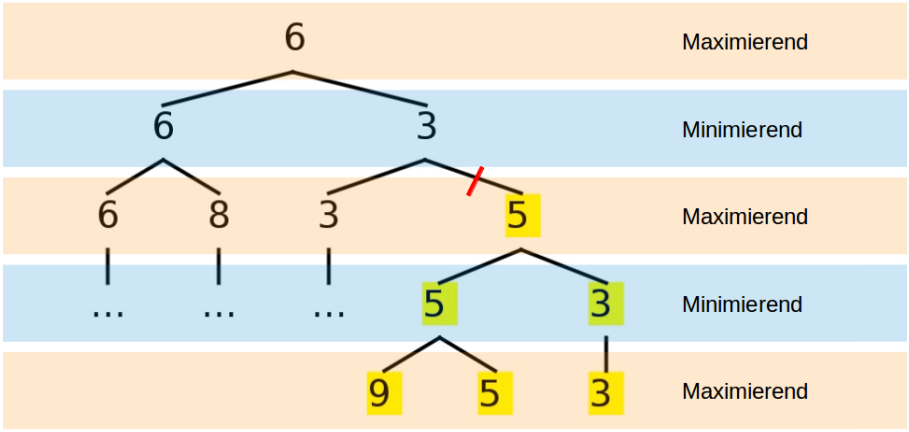


Abbildung 12: *Pruning-Möglichkeiten bei der Traversierung des Suchbaumes*

dieses nicht beachtet würde. Dementsprechend müssen Teilbäume, für welche die Bedingungen wahr sind, nicht ausgewertet werden.

Der Algorithmus des Alpha-Beta-Prunings erreicht immer den gleichen besten Halbzug, wie der Minimax-Algorithmus, unter Umständen jedoch in deutlich kürzerer Zeit, da Teilbäume für welche die Pruning-Bedingungen gelten nicht ausgewertet werden müssen.

Für die Effizienz des Algorithmus' spielt das Pruning die bedeutendste Rolle, wie der Name referenziert. Der Vorgang des Prunings von Teilbäumen kann dann erfolgen, wenn der beste oder ein sehr guter Schritt bereits gefunden wurde. Der Alpha-Beta-Algorithmus funktioniert demnach am besten, wenn der best-mögliche Schritt sich immer im linken äußeren Teilbaum befindet und alle weiteren Schritte absteigend bezüglich ihrer Wertigkeit sortiert sind.

Um dies zu gewährleisten findet für das Alpha-Beta-Pruning meist eine Form des Move-Ordering Anwendung. Weitere Informationen im Abschnitt 4.2 „Move-Ordering".

Mit einem Branching-Faktor b und einer Rekursionstiefe d gibt es zwei mögliche Varianten. Im ersten Fall findet ein aufsteigendes Move-Ordering statt (worst case!). In diesem Fall kann kein Pruning stattfinden, und der Algorithmus hat, wie zu erwarten, die gleiche Effizienz wie der Minimax-Algorithmus (3.1 „Minimax") von

$$O(b * b * b * \cdots * b) = O(b^d)$$

Im zweiten Fall gehen wir davon aus, dass ein perfektes Move-Ordering stattfindet und der beste Schritt immer zuerst gefunden wird. In diesem Fall müssen alle eigenen Züge untersucht werden. Dann allerdings genügt ein einziger gegnerischer Schritt (der Beste des Gegners), um alle weiteren Teilbäume zu prunen. Diese Möglichkeit bietet daher eine Komplexität von

$$O(b * 1 * b * 1 * \cdots * 1)$$

für eine ungerade Rekursionstiefe und

$$O(b * 1 * b * 1 * \cdots * b)$$

für eine gerade Rekursionstiefe. Also insgesamt eine Komplexität von:

$$O(b^{d/2}) = O(\sqrt{b^d})$$

[39] Bei perfektem Move-Ordering kann der Alpha-Beta-Pruning-Algorithmus also doppelt so tief suchen, wie der Minimax-Algorithmus.
Im Durchschnitt entspricht die Komplexität des Alpha-Beta-Algorithmus also:

$$O(b^{3d/4})$$

Um die genannte Funktionalität zu gewährleisten, arbeitet Alpha-Beta-Pruning nicht nur mit finalen Werten, sondern auch Grenzen. Diese beiden Grenzen sind:

α = der minimale Wert des besten Halbzuges, der bisher gefunden wurde für den maximierenden Spieler.

β = der maximale Wert des besten Halbzuges, der bisher gefunden wurde für den minimierenden Spieler.

Die Werte α und β bilden somit die Grenzen um den tatsächlichen Minimax-Wert. Für diesen ist garantiert, dass er immer zwischen diesen beiden Werten liegt.

Sobald man für den maximierenden Spieler eine Konstellation entdeckt, für die gilt, dass $\alpha >= \beta$, weiß man, dass ein Beta-Cut-Off erreicht wurde und die weiteren Teilbäume nicht weiter ausgewertet werden müssen, da ein berechneter Wert außerhalb der Grenzen von α und β liegt und somit keinerlei Relevanz für das Endergebnis haben kann. Wenn für den minimiernden Spieler ein $\alpha >= \beta$ gefunden wird, deutet dies auf ein Alpha-Cut-Off hin.

Eine Evaluierung eines Teilbaumes gibt demnach einen Wert zurück, der auf drei verschiedene Arten interpretiert werden kann. Das Zwischenergebnis kann entweder einen exakten Wert darstellen (der ein Cut-Off auf einer Ebene produziert) oder je nach Ebene (Maximierender/Minimierender Spieler) eine untere (Alpha) oder oberere (Beta) Grenze darstellen.

Diese Unterscheidung wird aber erst im Abschnitt 4.4 „Zobrist-Hash“ und 4.3 „Transposition Table“ relevant.

Der Pseudo-Code in der folgenden Abbildung trägt zum allgemeinen Verständnis bei:

```
alphabeta (TreeNode node, int depth, boolean maximizingPlayer, int alpha, int beta)

    if depth == 0
        return evaluierten Wert des aktuellen Board-Zustandes

    moveList = generateAllMoves()
    if maximizingPlayer
        bestValue = -INFINITY
        for (Move move : moveList)
            node = applyMove()
            // Recursion
            value = alphabeta (node, depth - 1, !maximizingPlayer, alpha, beta)
            // Saving best value
            if value > alpha
                bestValue = value
            // Maximizing alpha
            alpha = max (alpha, value)
            // Beta Cut-Off
            if alpha >= beta
                break
    else
        bestValue = INFINITY
        for (Move move : moveList)
            node = applyMove()
            // Recursion
            value = alphabeta (node, depth - 1, !maximizingPlayer, alpha, beta)
            // Saving best value
            if value < beta
                bestValue = value
            // Minimizing beta
            beta = min(beta, value)
            // Alpha Cut-Off
            if alpha >= beta
                break
    return bestValue
```

Abbildung 13: *Pseudo-Code für Alpha-Beta*

3.3 NegaMax

Der NegaMax Algorithmus gehört zur Familie der Minimax-Algorithmen und hat in seiner Grundform, wenn man die Implementierung außen vor lässt, eine identische Ausführungskomplexität wie der ursprüngliche Minimax-Algorithmus. Der grundlegende Unterschied zum Minimax-Algorithmus ist, dass davon ausgegangen wird, dass folgendes gilt:

$$\mathsf{max}(\mathsf{a}, \mathsf{b}) == -\mathsf{min}(-\mathsf{a}, -\mathsf{b})$$

Damit wird der Minimax-Algorithmus soweit vereinfacht, dass nicht mehr zwischen dem maximierenden und minimierenden Spieler unterschieden werden muss, um entweder das Maximum oder Minimum der entsprechenden Teilbäume zu ermitteln. Stattdessen kann das Ergebnis negiert und maximiert werden. Die Negierung führt dann automatisch dazu, dass abwechselnd minimiert und maximiert wird, sofern die Ausgangsannahme gilt.

Durch diese Vereinfachung erfolgt folgender Grundalgorithmus in Pseudo-Code:

```
negamax (TreeNode node, int depth)

    if depth == 0
        return evaluierten Wert des aktuellen Board-Zustandes

    moveList = generateAllMoves()

    for (Move move : moveList)
        node = applyMove()
        score = max (score, -negamax (node, depth - 1))

    return score
```

Abbildung 14: *Pseudo-Code für Negamax als Minimax-Erweiterung*

Die Grundannahme kann nicht nur auf den Minimax-Algorithmus übertragen werden, sondern auch auf das Alpha-Beta-Pruning. Die Veränderung des Alpha-Beta-Algorithmus ist entsprechend der Änderung des Minimax-Algorithmus und resultiert in folgendem Pseudocode:

```
negamax (TreeNode node, int depth, int alpha, int beta)

    if depth == 0
        return evaluierten Wert des aktuellen Board-Zustandes

    moveList = generateAllMoves()

    bestValue = -INFINITY
    for (Move move : moveList)
        node = applyMove()
        // Recursion
        value = -negamax (node, depth - 1, -beta, -alpha)
        // Saving best value
        if value > alpha
            bestValue = value
        // Maximizing alpha
        alpha = max (alpha, value)
        // Beta Cut-Off
        if alpha >= beta
            break

    return bestValue
```

Abbildung 15: *Pseudo-Code für Negamax als AlphaBeta-Erweiterung*

Wie bei dem rekursiven Aufruf zu erkennen ist, alternieren und negieren die Werte Alpha und Beta beim rekursiven Aufruf, um eine korrekte Minimierung und Maximierung auch bezüglich der Grenzen Alpha und Beta zu erreichen. In dieser Form als Erweiterung des AlphaBeta-Algorithmus entspricht die Zeit- und Raumkomplexität die des normalen AlphaBeta-Algorithmus'. Außer einer Vereinfachung des Codes bietet der NegaMax-Algorithmus so keinerlei Vorteile. Für weiterführende Algorithmen, wie zum Beispiel den NegaScout, bietet der NegaMax-Algorithmus jedoch eine Basis.

3.4 NegaScout

Der NegaScout-Algorithmus basiert auf der AlphaBeta Version des NegaMax-Algorithmus'. Er kann also als eine Erweiterung des Alpha-Beta-Pruning gesehen werden, der im schlechtesten Fall gleich viele Knoten wie das Alpha-Beta-Pruning traversiert und evaluiert. In einigen Fällen wird jedoch eine höhere Anzahl an Cut-Offs produziert, wodurch weniger Knoten ausgewertet werden müssen und den NegaScout-Algorithmus besser performen lässt, als ein einfacher AlphaBeta-Pruning-Algorithmus.

Die verbesserte Performance des NegaScout beruht auf der Annahme, dass der erste Teilbaum eines Knotens immer Teil der Principal Variation (siehe: 6 „Principal Variation“) ist und basiert somit maßgeblich auf einem guten Move-Ordering (siehe 4.2 „Move-Ordering“).

Das Grundprinzip des NegaScout-Algorithmus ist es, nur den ersten Teilbaum eines jeweiligen Knotens vollständig auszuwerten. Um Fehler und nicht perfektem Move-Ordering vorzubeugen, werden alle weiteren Teilbäume ausschließlich bezüglich ihrer möglichen Untergrenze (Alpha-Wert) überprüft. Das Überprüfen der Untergrenze entspricht keiner vollständigen Suche und dient ausschließlich dazu abzusichern, dass die Grundannahme richtig ist. Die Überprüfung der Untergrenze eines Teilbaumes erfolgt durch ein sogenannten Null-Window-Search.

Hierbei werden bei dem rekursiven Aufruf Alpha und Beta so gesetzt, dass diese direkt nebeneinander liegen. Dadurch wird immer ein Wert erzeugt, welcher unter Alpha oder über Beta liegt. Anhand der Form des erzeugten Cut-Offs können die Grenzen der weiteren Suche angepasst werden oder es kann effizient bestätigt werden, dass der erzeugte Wert nicht über einer festgelegten Grenze liegt.

Im Fall des NegaScout-Algorithmus wird mit Hilfe des Null-Window-Search ausschließlich sichergestellt, dass der Wert des Teilbaumes (alle außer dem ersten) nicht über dem Alpha-Wert liegt. Ist dies der Fall, stellt der neue Teilbaum keine mögliche Verbesserung des Gesamtergebnisses dar, und die Annahme, dass sich der erste Teilbaum in der Principal Variation befindet, ist soweit bestätigt.

Der rekursive Aufruf des Null-Window-Searches erfolgt mit

$$\mathsf{Alpha} = -\mathsf{Alpha} - 1$$

und

$$\mathsf{Beta} = -\mathsf{Alpha}$$

Wenn der Null-Window-Search jedoch über dem Alpha-Wert liegt, ist die Ursprungsannahme falsch, und der entsprechende Teilbaum muss normal traversiert werden (normaler rekursiver Aufruf des NegaScout-Algorithmus).

Folglich muss der NegaScout-Algorithmus niemals mehr Knoten untersuchen, als der Alpha-Beta-Algorithmus. Im Falle eines schlechten Move-Orderings allerdings müssen mehrere Teilbäume mehrfach traversiert werden, einmal als Null-Window-Search, und wenn die Überprüfung fehlschlägt, noch einmal vollständig. Dadurch ist der NegaScout-Algorithmus bei zufälligem oder schlechten Move-Ordering durch den Overhead der mehrfach traversierten Teilbäume langsamer, als das normale Alpha-Beta-Pruning.

Eine Implementierung des NegaScout-Algorithmus ist als Pseudocode in folgender Grafik abgebildet:

```
negascout (TreeNode node, int depth, int alpha, int beta)

    if depth == 0
        return evaluierten Wert des aktuellen Board-Zustandes

    moveList = generateAllMoves()

    for (Move move : moveList)

        node = applyMove()

        // First Move as supposed Principal Variation
        if (move == firstMove)
            bestValue = -negascout (node, depth - 1, -beta, -alpha)
        else
            // Null-Window-Search
            bestValue = -negascout (node, depth - 1, -alpha - 1, -alpha)
            // Check on the lower bound
            if value > alpha && value < beta
                // Normal full Search
                bestValue = -negascout (node, depth - 1, -beta, -alpha)

        // Maximizing alpha
        alpha = max (alpha, bestValue)
        // Beta Cut-Off
        if alpha >= beta
            break

    return bestValue
```

Abbildung 16: *Pseudo-Code für NegaScout*

3.5 Iterative Deepening

Iterative Deepening bietet ein Framework für Depth-First-Search-Algorithmen. Genutzt und entwickelt wurde der Algorithmus als Basis für sinnvolles Zeitmanagement bei der Traversierung eines Zustandsbaumes.

Der Algorithmus arbeitet, indem er iterativ seine Rekursionstiefe erhöht, bis die maximale gewünschte Rekursionstiefe erreicht ist. Dabei wird ein Teilbaum in geringerer Rekursionstiefe mehrfach traversiert. In vielen Fällen bedient sich der Iterative Deepening-Algorithmus eines Depth-First-Search Algorithmus' für die Traversierung der Teilbäume.

Iterative Deepening bietet im Verhältnis zu normalen Depth-First-Search-Algorithmen einige nicht zu unterschätzende Vorteile. In vielen Fällen ist die Zeit, die zur Berechnung des jeweiligen besten Zuges notwenig ist, begrenzt. So muss die Berechnung und Traversierung meist durch Zeitnot unterbrochen werden. Bei der Nutzung eines klassischen Minimax oder Alpha-Beta-Algorithmus ist der Baum nicht symmentrisch traversiert worden, und es ist möglich, dass ein guter oder der beste Zug nicht gefunden wurde, da er sich in der rechten Hälfte des Baumes befindet.

Im Gegensatz dazu hat der Iterative Deepening-Algorithmus zwar nicht seine volle Rekursionstiefe erreicht, aber alle Möglichkeiten eines Zuges wur-

den bis zu einer gleich tiefen Rekursionstiefe evaluiert. Somit wird bei Iterative Deepening immer ein Zug ausgewählt, der bis zu der erreichten Rekursionstiefe das beste Ergebnis geliefert hat. Eine klassische Version des Iterative Deepening lässt sich auf folgende Weise implementieren:

```
iterativeDeepening(TreeNode node)

    for (depth = 0; depth <= FINAL_DEPTH; depth++)

        bestMove = depthFirstSearchAlgorithm (board, depth)

        if (timeIsUp)
            break

    return bestMove
```

Abbildung 17: *Pseudo-Code für Iterative Deepening*

Die normale Version des Iterative Deepening bietet hauptsächlich den Vorteil des guten Zeitmanagements mit guten Zwischenergebnissen, wenn die Zeit nicht reicht für eine vollständige Traversierung bis zu einer gewünschten Rekursionstiefe. Ein Nachteil der klassischen Version ist unverkennbar: dass alle Knoten geringerer Rekursionstiefe mehrfach traversiert werden, wodurch ein Overhead entsteht. Das Iterative Deepening Framework kann mit beliebigen Depth-First-Search Algorithmen genutzt werden.

Die Anzahl an traversierten Knoten ist folgende:

$$\#\mathsf{Knoten} = (\mathsf{d})\mathsf{b}^1 + (\mathsf{d}-1)\mathsf{b}^2 + (\mathsf{d}-2)\mathsf{b}^3 + \cdots + (1)\mathsf{b}^\mathsf{d}$$

Dies ergibt eine Zeit-Komplexität von:

$$\mathsf{O}(\mathsf{b}^\mathsf{d})$$

und entspricht der von Breadth-First-Search Algorithmen.

In einem direkten Vergleich der Anzahl tatsächlich traversierter Knoten zwischen Iterative Deepening Search (IDS) und Breadth-First-Search (BFS) wird deutlich, das der Overhead durch mehrfach traversierte Knoten sehr gering ausfällt. Sei beispielsweise die Rekursionstiefe $d = 5$ und der Branching-Faktor $b = 10$, so berechnet sich die Anzahl traversierter Knoten auf folgende Weise [43]:

$$\#\mathsf{Knoten}(\mathsf{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123.450$$

$$\#\mathsf{Knoten}(\mathsf{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111.110$$

3.6 Iterative Deepening mit Transposition Table und Move-Ordering

Sobald der klassische Iterative Deepening mit einer Form der Memory-Nutzung versehen wird, wie beispielsweise Transposition Tables (siehe: 4.3 „Transposition Table“), erhöht sich die Effizienz und der genannte Overhead durch mehrfache Traversierung der Teilbäume entfällt.

Mit Hilfe von Transposition Tables wird jeder evaluierte Knoten in einer globalen Tabelle registriert. Bei iterativer Rekursionstiefe sind die Zwischenergebnisse von Knoten geringerer Rekursionstiefe schon bekannt. Dadurch kann bei jeder Rekursionstiefe ein Move-Ordering erzeugt werden, welches bis zu der erreichten Rekursionstiefe perfekt ist und ausschließlich auf den bereits evaluierten Ergebnissen beruht. Das Move-Ordering selber ist extrem effizient, da ausschließlich existente Werte des Transposition Tables genutzt werden.

Depth-First-Search-Algorithmen, welche in das Iterative Deepening Framework integriert, sind können nun von dem extrem guten Move-Ordering profitieren. Bei Nutzung beispielsweise eines Alpha-Beta oder NegaScout Algorithmus' kann eine sehr große Anzahl an Cut-Offs erzeugt werden, wodurch in späteren Rekursionen nur noch Bruchteile des gesamten Zustandsbaumes evaluiert werden müssen, da der beste Schritt mit hoher Wahrscheinlichkeit im ersten Teilbaum zu finden ist.

Trotz des vermeindlichen Overheads durch mehrfache Traversierung des Baumes bietet ein Iterative Deepening, welches Transposition Tables und Move-Ordering nutzt, in der Praxis eine schnellere und effizientere Traversierung des gesamten Baumes bis zu einer finalen Rekursionstiefe, als ein Alpha-Beta-Algorithmus mit Transposition Tables und Move-Ordering. Im Kapitel „Tests“ werden die Unterschiede und die Performanz der einzelnen Algorithmen zueinander weiter verdeutlicht.

3.7 MTD(f)

Der MTD(f)-Algorithmus gehört zu der Familie der Minimax-Algorithmen und misst sich bezüglich seiner Effizienz und Geschwindigkeit mit NegaScout (siehe: 3.4 „NegaScout“), wobei eine Reihe an Tests ergeben hat, dass MTD(f) bei guten Voraussetzungen besser und schneller arbeitet, als andere Minimax-Algorithmen, wie NegaScout [36].

Im Gegensatz zu den bisher behandelten Minimax-Algorithmen arbeitet der MTD(f)-Algorithmus ausschließlich mit wiederholten rekursiven Null-Window-Aufrufen, welche bei jedem Aufruf Cut-Offs erzeugen, um eine mögliche obere oder untere Grenze (Alpha oder Beta) zu checken. Dabei wird der Umstand ausgenutzt, dass gerade diese Form der Baumtraversierung besonders effizient und schnell ausführbar ist.

Das Prinzip hinter MTD(f) ist es, ausschließlich die untere und obere Grenze des Minimax-Wertes aneinander anzunähern, bis der exakte Minimax-Wert gefunden wurde. Normalerweise werden die Grenzen Alpha und Beta so gesetzt, dass der erwartete Wert immer zwischen ihnen liegt, bis dieser gefunden wurde. MTD(f) macht das genaue Gegenteil. Jeder rekursive Aufruf ergibt einen Wert, der immer außerhalb der gesetzten Grenzen liegt. So kann der Minimax-Wert von außen angenähert werden.

Typischerweise wird der MTD(f)-Algorithmus in einem Iterative Deepenig Framework eingebunden und ruft eine Form einer Alpha-Beta Implementierung auf. Der Initiale Aufruf des MTD(f) beinhaltet immer auch einen Integer-Wert *f*. Dies ist der so genannte *first-guess*, ein geschätzter Wert, der von der Auswertung des Gesamtbaumes erwartet wird.

Je besser der geschätzte Wert ist, desto weniger Aufrufe von MTD(f) müssen getätigt werden. Im Idealfall, wenn der first-guess dem exakten Minimax-Wert entspricht, werden nur zwei Aufrufe von MTD(f) getätigt, einen, um die Obergrenze zu setzen, und einen für die Untergrenze.

Im Normalfall, wenn MTD(f) in einem Iterative Deepening Framework aufgerufen wird, kann man das Ergebnis von MTD(f) einer geringeren Rekursionstiefe nehmen und MTD(f) für eine erneute, tiefere Suche übergeben. Abhängig davon, wie stark oszillierend die Werte abhängig von ihrere Rekursionstiefe sind, fällt das Gesamtergebnis gut oder weniger gut aus.

Weitere Taktiken sind auch, den first-guess des letzten Aufrufs der gleichen Spielerfarbe zu verwenden oder absichtlich zu hoch und zu tief zu schätzen, um danach eine bessere Einschätzung für den first-guess zu bekommen.

Ähnlich wie auch der NegaScout Algorithmus entfaltet der MTD(f) erst dann seine ganze Effizienz, wenn der verwendete Depth-First-Search Algorithmus eine Form von Memory-Nutzung beinhaltet, die ein erneutes Auswerten aller Knoten erübrigt. Besonders durch den Umstand, dass eine Rekursion unter Umständen, bei zum Beispiel starker Oszillation der Werte, sehr oft aufgerufen wird, ist eine Nutzung von beispielsweise Transposition Tables unabdingbar, um den MTD(f)-Algorithmus wettbewerbsfähig zu machen.

In folgender Grafik ist der Pseudocode eines typischen MTD(f) Algorithmus aufgezeigt:

```
mtdf(TreeNode node, int firstguess, int depth)

    guess = firstguess
    upperBound = INFINITY
    lowerBound = -INFINITY

    // closing the bounds toward each other
    while lowerBound < upperBound

        // setting a better guess for the recursion
        if guess == lowerBound
            beta = guess + 1
        else
            beta = guess

        // recursive null-window call to alphaBeta
        guess = alphaBeta (node, beta - 1, depth)

        // setting the bounds depending on the previous result
        if guess < beta
            upperbound = guess
        else
            lowerbound = guess

    return guess
```

Abbildung 18: *Pseudo-Code für MTD(f)*

Der Aufruf des MTD(f) erfolgt normalerweise in einem Iterative Deepening Framework, welches in folgender Grafik als Pseudocode abgebildet ist:

```
iterativeDeepening(TreeNode node)

    firstguess = 0

    for (depth = 0; depth <= FINAL_DEPTH; depth++)

        firstguess = mtdf (board, firstguess, depth)

        if (timeIsUp)
            break

    return firstguess
```

Abbildung 19: *Pseudo-Code für ein Iterative Deepening Framework für MTD(f)*

4 Algorithmische Erweiterungen

Die Grundalgorithmen selber bilden eine Basis für die Traversierung des Zustandsbaumes. Darauf können algorithmische Erweiterungen und weitere Komponenten aufbauen, welche zu einer effizienteren und schnelleren Suche führen können.

4.1 Dreifache Wiederholung eines Spielzustandes

Eine Erweiterung der normalen Suchalgorithmen, welche spielendscheidend sein kann, ist die der dreifachen Wiederholung eines dagewesenen Spielzustands. Eine dreifach wiederholte Spielsituation führt zum Verlieren desjenigen Spielers, der sie zum dritten Mal herbeigeführt hat.

Die Wiederholung muss nicht nur über eine Suche, sondern über das gesamte Spiel gespeichert und abgerufen werden. Somit entfällt die Möglichkeit, sofern Transposition Tables (siehe: 4.3 „Transposition Table“) implementiert wurden, die Zobrist-Hashing-Funktion dieser zu nutzen, da der Hash sich bei jeder Suche ändert.

Die nächst einfachere Möglichkeit bildet eine globale Instanz eines Zobrist-Hash-Generators, welcher, unabhängig vom Spielfortschritt, den gleichen Hash für eine identische Spielfeldsituation berechnet. Die Hashes werden dann in einer Liste global abgelegt und bei jeder Suche abgeglichen.

4.2 Move-Ordering

Für bestimmte Algorithmen (siehe beispielsweise: 3.2 „Alpha Beta Pruning“) ergibt sich eine besonders hohe Effizienz, wenn der beste Schritt immer zuerst ausgewertet wird, da eine hohe Anzahl an Cut-Offs erzeugt werden können und viele der weiteren Teilbäume nicht evaluiert werden müssen. Um dies zu erreichen, wird vor den rekursiven Aufrufen der Teilbäume des jeweiligen Algorithmus' ein Move-Ordering ausgeführt, welches versucht, die Kindknoten abhängig von ihren approximierten Endergebnissen zu sortieren.

4.2.1 Move-Ordering mittels eines Minimax-Algorithmus

Eine mögliche Form des Move-Ordering ist es, eine weitere, eigenständige und flachere Suche für jeden Knoten zu erzeugen und den jeweils flachen Baum (meist 0-2 Rekursionstiefen) eigenständig auszuwerten. Mithilfe dieser Zwischenergebnisse kann dann eine Sortierung der eigentlichen Kindknoten erfolgen.

Wie zu erkennen ist, produziert das Move-Ordering einen konstanten Overhead, der bei jeder Rekursion und für jeden Teilbaum erneut auftritt. Da aber durch das Move-Ordering unter Umständen ganze Teilbäume nicht mehr ausgewertet werden müssen, überwiegen meist die Vorteile des Verfahrens. Bei dieser Form des Move-Ordering kann jeder beliebige Grund-Such-Algorithmus eingesetzt werden.

4.2.2 Move-Ordering mit Memory-Enhancement

In bestimmten Situationen kann ein Move-Ordering komplett ohne eine eigenständige Suche auskommen und dadurch extrem effizient und genau werden. Eine Möglichkeit besteht daraus, die Zwischenergebnisse des Move-Ordering ausschließlich aus bereits evaluierten und ausgewerteten Spielfeldzuständen zu beziehen und so eine Neuberechnung überflüssig zu machen.

Diese Form des Move-Ordering ist jedoch an bestimmte Voraussetzungen geknüpft. Jeder Knoten, bei denen ein Move-Ordering stattfinden soll, muss im Vorfeld der Suche bereits evaluiert worden sein. Dies ist fast ausschließlich bei der Nutzung eines Iterative Deepening Framework (siehe: 3.6 „Iterative Deepening mit Transposition Table und Move-Ordering“) der Fall. Außerdem ist die Nutzung einer Memory, wie Transposition Tabes, notwendig, um evaluierte Boardzustände für Lookups bereit zu halten.

Sind diese Voraussetzungen erfüllt, kann man bei jedem Knoten, bei dem ein Move-Ordering geplant ist, davon ausgehen, dass er bereits evaluiert wurde. Ein Lookup in das Transposition Table ist daher immer erfolgreich. Es bedarf nun eines einfachen Lookups jedes Kindknotens und einer Sortierung dieser anhand der Werte aus dem Transposition Table, um ein schnelles und gutes Move-Ordering zu erreichen.

4.3 Transposition Table

In vielen Brettspielen wie Arimaa und auch Schach kann es vorkommen, dass eine gleiche Spielfeldsitation über eine Abfolge verschiedener einzelner Schritte erreicht werden kann. Bei geringer Rekursionstiefe spielt es oft keine große Rolle, die Situationen mehrfach auszuwerten. Bei einer höheren Rekursionstiefe allerdings sind davon ganze Teilbäume betroffen (die an unterschiedlichen Stellen im Suchbaum die gleiche Situation abbilden und somit von dort an identisch sind). Das Erreichen gleicher Positionen mit unterschiedlichen Schritten nennt sich 'transposing' [30].

Eine Möglichkeit, dem vorzubeugen und Situationen nur ein einziges Mal auszuwerten, bieten die Transposition Tables. Hierbei handelt es sich lediglich um eine Hashmap, in der eine Spielfeldsituation mit weiteren Parametern verzeichnet ist. In jedem Halbzug kann nun getestet werden, ob die aktuelle Spielfeldsituation bereits evaluiert wurde und ob eine weitere Evaluation von Nöten ist.

Die Spielfeldaufstellung wird üblicherweise als Schlüssel in der Hash-Map verwendet. Eine elegante, extrem effiziente und schnelle Lösung dazu bietet das Zobrist-Hashing (siehe Abschnitt 4.4 „Zobrist-Hash“).
Elemente, die in der Hashmap gespeichert werden sind folgende:

- *hash* - der Hash-Wert der Spielfeldsituation
- *depth* - die Rekursionstiefe, in welcher der Teilbaum evaluiert wurde
- *score* - der evaluierte Wert des Teilbaumes
- *nodeType* - der Typ des Knotens. Es kann exakt, alpha oder beta sein.

Die gespeicherte Variable *depth* ist sehr wichtig, um zu testen, in welcher Rekursionstiefe der Teilbaum evaluiert wurde. So kann es sein, dass die gleiche Spielfeldsituation einmal im Baum nach der zweiten Rekursion und einmal nach der sechsten Rekursion auftritt.

Es ist wichtig, dass der evaluierte Knoten in dem Transposition Table sich näher an dem Root-Knoten befindet, um mindestens die gewünschte Gesamt-Rekursionstiefe zu erreichen. Ansonsten ist es möglich, dass dem Knoten mit Rekursionstiefe $d = 2$ der Wert des Knotens an $d = 6$ zugeordnet wird. Bei einer gesamten gewünschten Rekursionstiefe von $d = 8$ wurde der Teilbaum mit $d = 2$ durch den Eintrag im Transposition Table effektiv nur bis $d = 4$ ausgewertet.

Wenn wir die gleiche Situation anders herum betrachten und würden dem Knoten an $d = 6$ den Wert des Knotens an $d = 2$ zuordnen, haben wir effektiv lokal bis zu einer Rekursionstiefe von $d = 12$ traversiert und evaluiert. Die gespeicherte Variable *score* beinhaltet, wie zu erwarten, den evaluierten Wert des Teilbaumes mit dem Typ der gespeicherten Variable *nodeType*.

Die gespeicherte Variable *nodeTyp*e gibt einen Hinweis darauf, um welchen Typ Knoten es sich bei der gespeicherten Spielfeldsituation handelt. Hier gibt es verschiedene Möglichkeiten. Der einfachste Fall ist ein exakter Wert, welcher den exakten Wert des Teilbaumes zurückgibt. Im Falle eines Alpha-Beta-Cut-Offs kann dies jedoch nicht als exakter Wert interpretiert werden. NodeType kann also nicht nur exakte Werte zurückgeben, sondern auch einen sogenannten *alpha* oder *beta*-Wert. Diese können dann später zwar nicht genutzt werden, um einen Teilbaum komplett nicht mehr auswerten zu müssen, aber unter Umständen, um das Fenster zwischen den übergebenden Werten des Alpha-Beta-Pruning (Alpha und Beta) weiter einzugrenzen und so eine bessere Suche zu gewährleisten und unter Umständen weitere Cut-Offs zu erzeugen.

Der Schlüssel beziehungsweise *hash*, der Hashmap unter der Spielfeldsituationen gespeichert werden, ist 64bit lang. Dies gibt eine extrem große Anzahl zu speichernder Möglichkeiten. Allerdings sind auch hier Kollisionen nicht grundsätzlich vermeidbar. Für Spiele wie Arimaa oder Schach werden 64 bit als Schlüssel aber meist als ausreichend angesehen, um eine möglichst hohe Performace zu erhalten. Die Möglichkeit von Kollisionen und folgende negative Effekte dieser sind jedoch selten genug, dass sie nicht als praktisches Problem gelten [1].

Es gibt die Möglichkeit, ein globales Transposition Table über das gesamte Spiel über zu führen. Dies bedingt aber, dass mit der Zeit alle Einträge der Hashmap durch neuere ersetzt werden müssen. Meist reicht ein Transposition Table, welches sich mit der Berechnung des jeweiligen Halbzuges beschäftigt.

4.4 Zobrist-Hash

Das Zobrist-Hashing bietet eine Möglichkeit, einen (fast) eindeutigen Hash aus einer Spielfeldsituation zu erzeugen, um diesen Hash als Schlüssel zu einer Hashmap verwenden zu können. Hintergrund dieses Verfahrens sind die Transposition Tables (siehe 4.3 „Transposition Table“) und die Möglichkeit, eine Boardsituation speichereffizient und schnell speichern zu können. Außerdem soll gegeben sein, dass bei einer Änderung des Spielfeldes der neue Schlüssel schnell und effizient erzeugt werden kann, ohne diesen jedes Mal neu berechnen zu müssen, um die Effizienz zu steigern. Bei dem zu berechnenden Schlüssel handelt es sich um einen 64 bit Integer.

Für die eigentliche Berechnung des Zobrist-Hashes werden zunächst für alle Kombinationen der Farbe, Figurtyp und Feldposition ein eindeutiger zufälliger 64 bit Integer erzeugt. Bei zwei Farben, 6 Figurtypen und 64 möglichen Spielfeldfeldern ergibt das

$$2 * 6 * 64 = 768$$

verschiedene eindeutige zufällige 64 bit Werte.
Um für eine beliebige Boardsituation einen eindeutigen Zobrist-Hashwert zu berechnen, eignet sich folgender Algorithmus:

```
zobrist (TreeNode board)
    hash = 0
    for (i = 0; i < 64; i++)
        if positionNotEmpty (i)
            Piece piece = getPieceAt (i)
            hash = hash XOR table[piece.color][piece.type][i]
    return hash
```

Abbildung 20: *Berechnung eines Zobrist-Hashes*

Hierbei wird also ein 64 bit Zobrist-Hash durch eine XOR-Verknüpfung mit allen vorhandenen Figuren auf dem Board beziehungsweise den festgelegten zufälligen 64-bit-Integern erzeugt. Die Berechnung für jeden einzelnen Knoten des Suchbaumes erneut auszuführen wäre ineffizient.

Stattdessen kann man die Möglichkeit, dass für eine XOR-Verknüpfung das Idempotenz-Gesetz und das Kommutativ-Gesetz gelten, ausnutzen. Das Idempotenz-Gesetz besagt, dass eine Menge mit sich selber mittels XOR verknüpft die leere Menge bildet.

Aus diesen Grundsätzen kann man schließen, dass das Setzen einer Figur auf dem Spielfeld gleichkommt wie den vorhandenen Hashwert mittels des XOR-Operators mit dem festgelegten zufälligen 64-bit-Integers zu verknüpfen. Genauso kann das Entfernen einer Figur mittels einer weiteren XOR-Verknüpfung mit dem selben Integer erfolgen.

Weitergehend kann man also einen Schritt auf dem Spielfeld mit nur zwei XOR-Verknüpfungen auf den Zobrist-Hash abbilden und einen neuen validen Hash erzeugen, welcher der neuen Spielfeldsituation entspricht.
Ein Beispiel mit folgender Spielfeldsituation:

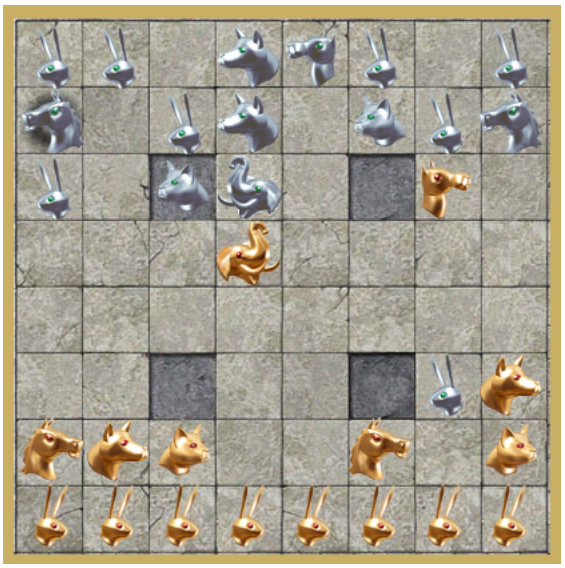


Abbildung 21: *Beispiel einer Spielfeldsituation*

Jetzt wird der folgende Schritt ausgeführt *Ed5e*. Das bedeutet, dass der Goldene Elefant von d5 auf e5 bewegt werden soll.
Für die Neuberechnung des *hash*-Wertes bedeutet das folgende Befehle:

hash = hash XOR table[GOLD][ELEPHANT][36]
hash = hash XOR table[GOLD][ELEPHANT][37]

Die erste XOR-Verknüpfung entfernt den goldenen Elefanten von d5, während die zweite XOR-Verknüpfung den goldenen Elefanten auf e5 setzt.

4.5 Quiescence-Search

Die Quiescence-Search ist die Bezeichung einer limitierten, positionell und konzeptionell angewandten Erweiterung und Vertiefung der Rekursionstiefe bei der Traversierung des Zustandsbaumes. Quiescence-Search ist unter anderem ein Lösungsansatz, um dem sogenannten Horizon-Effect vorzubeugen.

Der Horizon-Effect resultiert auf dem Umstand, dass bei einer Traversierung des Suchbaumes nur bis zu einer bestimmten, meist hart kodierten, Rekursionstiefe gerechnet werden kann. Bei einer Evaluierung des jeweiligen Zustandes kann es sein, dass ein gutes Ergebnis erzielt wurde, da beispielsweise ein Hund des Gegners gefangengenommen werden konnte. Der Horizon-Effect tritt auf, wenn in dem nächsten Halbzug jedoch das eigene Kamel gefangengenommen werden würde. Die gewählte Principal Variation (siehe: 6 „Principal Variation“) des Suchbaumes führt nun zu einem sehr schlechten Ergebnis, da schlussendlich ein Hund-Kamel-Handel zustande kam. Diese Begrenzung des Sichtfeldes durch die gewählte Rekursionstiefe nennt sich Horizon-Effect.

Eine Quiescence-Search analysiert an möglichen Endknoten mit einer Rekursionstiefe d = 0 nun das Spielfeld und erhöht bei Bedarf und unter bestimmten Voraussetzungen die Rekursionstiefe des aktuellen Knotens.

Hierdurch wird an bestimmten, vielversprechenden Positionen eine vertiefte Suche ausgeführt, um sicherzustellen, dass nicht bedingt durch den Horizon-Effect eine schlechtere Lösung gewählt wird.

Ein weiterer spezieller Einsatzbereich einer situationsbedingten Vertiefung der Rekursionstiefe ist bei einer Trapkontrolle möglich, um eine maximale Sicherheit bei einer Verteidigung und einem Angriff zu gewährleisten. Hier müssen aber sehr spezielle und selten vorkommende Rahmenbedingungen gelten, um eine Suche nicht unnötig zu verlangsamen.

Der mit-wichtigste Grund einer Quiescence-Search ist das Finden von Gewinnzügen. Gerade im Endspiel ist eine Positionierung und die Bedrohung eines Hasen, die Endlinie zu erreichen, von extrem hoher Bedeutung. Hier ist es sehr ratsam, zum Beispiel für weit vorgerückte Hasen eine erweiterte Suche anzuwenden, um möglicherweise einen Halbzug zu finden, der in 3 oder 4 Zügen zu einem Forced-Win führen kann, welcher außerhalb des Horizontes der eigentlichen Suche liegt.

4.6 Parallelisierung

Durch den hohen Branching-Faktor (siehe: 5 „Branching-Faktor“) und die Menge an möglichen Zügen jedes Spielers in Arimaa ist eine Traversierung des Suchbaumes sehr zeitaufwendig und langsam. Auch durch ein hohes Maß an Pruning-Möglichkeiten, die verschiedene Formen von Grundalgorithmen zur Verfügung stellen und situationsbedingte Vertiefungen der Rekursionstiefe durch Quiescence-Search ist die Traversierung sehr zeit- und rechenintensiv.

Es gibt verschiedene Möglichkeiten, die Traversierung und Evaluierung des Suchbaumes durch verschiedene Prozesse oder Threads darzustellen und somit eine Parallelisierung der Berechnung zu erreichen. Doch der Umstand, dass bei Minimax-Algorithmen die Berechnung eines Knotens das Ergebnis seiner Kindknoten bedingt, erschwert eine ideale Parallelisierung. Meist entsteht ein hohes Maß mehrfacher Berechnungen und Overhead durch nötige Kommunikation der Threads.

Doch gerade bei wettbewerbstauglichen Programmen ist eine Parallelisierung der eigentlichen Traversierung des Suchbaumes unabdingbar.

4.6.1 Shared Hash Table

Eine Parallelisierung über das sogenannte *Shared Hash Table*-Prinzip ist die einfachste Form einer Implementierung und Idee für die Parallelisierung von Suchbäumen. Das Prinzip von Shared Hash Tables ist es, allen Threads den gleichen Wurzelknoten des Baumes zu übermitteln und alle Threads zu starten. Dabei ist es zwingend erforderlich, dass alle Threads eine gleiche Instanz von Hash Tables oder Transposition Tables nutzen, da erst durch die Memory-Nutzung eine Parallelisierung zu Stande kommt.

Die eigentliche Parallelisierung der Traversierung des Baumes erfolgt durch den Umstand des Nichtdeterminismus, mit welchem das Betriebssystem die Threads steuert und kontrolliert. Im Wurzelknoten bearbeiten noch alle Threads den gleichen Knoten. Mit jedem erzeugten Eintrag eines untersuchten und evaluierten Knotens in das Transposition Table wird eine Verteilung der einzelnen

Threads vergrößert, da bereits evaluierte Knoten so nicht noch einmal ausgewertet werden müssen.

Die Effizienz der Parallelisierung basiert also auf dem Umstand, dass bei einer Traversierung des Baumes Einträge anderer Threads in das Transposition Table genutzt werden können, um selbst Cut-Offs zu erzeugen und die eigene Suche zu verkürzen. Je tiefer die Suche also voranschreitet, desto mehr verteilen sich die einzelnen Threads auf den Baum und desto mehr parallelisiert sich die Suche.

Im Verhältnis zu erweiterten und komplexeren Algorithmen der Parallelisierung, wie sie in den folgenden Abschnitten zu finden sind, hat die Parallelisierung einen geringen Geschwindigkeitesvorteil. Die Skalierung, betrachtet als Nodes-Per-Second, im Bezug auf die Nutzung multipler Prozessoren ist jedoch fast perfekt [24].

4.6.2 YBWC - Young Brothers Wait Concept

Das Young Brother Wait Concept ist eines der Grundkonzepte der Parallelisierung von Depth-First-Search-Algorithmen wie Alpha-Beta. Der Algorithmus wertet dabei den Baum des allerersten Kindknoten vollständig aus, bevor die anderen Teilbäume parallel abgearbeitet werden. Das hinter dem Algorithmus stehende Prinzip ist es, durch eine vollständige Auswertung des ersten Teilbaumes, einen guten ersten Alpha-Wert zu schaffen, der eine hohe Anzahl an Cut-Offs vor einer weiteren Evaluierung der restlichen Knoten erzeugt und somit grundsätzlich die Menge der auszuwertenden Teilbäume reduziert.

Eine gleichmäßige Auslastung der Prozessoren während der eigentlichen parallelisierten Auswertung ist essentiell für eine gute Parallelisierung und die Nutzung des Young Brother Wait Concepts. Die Parallelisierung erfolgt für alle Teilbäume eines jeden Knotens im Suchbaum, unabhängig, ob dieser sich in der Principal Variation befindet oder nicht. Generell gesehen, und im Gegensatz zu erweiterten Algorithmen, wie Dynamic Principal Variation Splitting (In dieser Ausarbeitung nicht behandelt), ist ein Prozess alleine und vollständig verantwortlich für die Auswertung und Rückgabe des evaluierten Wertes eines Knotens, sobald er diesen betritt. Eine Übertragung des Knotens an einen anderen Prozess ist nicht vorgesehen und möglich in der normalen Ausprägung des YBWC. Normalerweise haben ein Knoten und alle seine Teilbäume den gleichen verantwortlichen Prozess. Wenn dies nicht der Fall sein sollte, muss das Ergebnis zwischen den Prozessen kommuniziert werden.

In dem Moment, wo ein Prozess P1 seine Aufgabe vollständig erledigt und einen möglichen Teilbaum traversiert und ausgewertet hat, wählt er einen beliebigen Prozess P2 aus und schickt diesem eine *Request-Work*-Nachricht. Wenn der Empfänger-Prozess keine Arbeit zur Verfügung hat, leitet er die Nachricht an einen beliebigen Prozess weiter. Die Weiterleitungen sind durch eine festgelegte maximale Anzahl an Zwischenstationen begrenzt. Ist diese überschritten, wird die Nachricht verworfen und muss von P1 erneut geschickt werden.

Der Prozess P2 hat Arbeit, wenn folgende Kriterien erfüllt wurden:

- P2 gehört ein Knoten
- Der erste (PV)-Knoten der Kinder wurde vollständig ausgewertet

Der erste Knoten der direkten Teilbäume des Knotens von P2, welcher die Kriterien erfüllt, wird nun zu einem *split-point*. An diesem *split-point* wird eine weitere parallele Suche gestartet.

Sobald die weitere parallele Verarbeitung an dem *split-point* startet, besteht nun eine Master-Slave-Beziehung zwischen P2 und P1. Wenn entweder die gesamte Arbeit der Traversierung eines Knotens vollendet wurde oder ein Slave-Prozess innerhalb eines Teilbaumes eine Cut-Off-Möglichkeit des obersten Knotens von P2 gefunden hat, gehen die Slave-Prozesse in einen Warte- oder Leerlauf-Zustand über. Wenn der Master-Prozess P2 seine Knoten fertig bearbeitet hat und keine weitere Arbeit mehr verfügbar ist, ordnet er sich als Slave ein und sendet eine *Request-Work*-Nachricht an einen beliebigen Slave-Prozess, um dessen Arbeit zu beschleunigen.

Die Nutzung des YBWC hat weitreichende Veränderung eines grundlegenden, beispielsweise Iterative Deepening, Algorithmus zur Folge, da bei einer Erzeugung von Cut-Offs in Teilbäumen Sprünge realisiert werden müssen, um eine überflüssige Auswertung von abgeschnittenen Teilbäumen zu unterbinden. Für die Sprünge und verbesserte Alpha- und Beta-Werte müssen meist Arrays genutzt werden, um allen Slave-Prozessen einer bestimmten Rekursionstiefe die Informationen zu übermitteln, die gegebenenfalls Auswirkung auf die Arbeit dieser haben (veränderte Alpha, Beta-Werte und Cut-Offs auf bestimmten Rekursionstiefen).

4.6.3 ABDADA - Alpha-Bêta Distribué avec Droit d'Anesse

Der Name ABDADA kommt aus dem Französischen und bedeutet soviel, wie *Distributed Alpha-Beta Search with Eldest Son Right*. Der Algorithmus basiert auf dem YBWC-Algorithmus in der Form, dass eine Auswertung der Geschwisterknoten erst dann geschieht, wenn der erste Teilbaum vollständig ausgewertet wurde.

Während der YBWC-Algorithmus aber weitreichende Änderungen des Such-Algorithmus erfordert, kommt ABDADA fast komplett ohne diese aus. Der Grund hierfür ist, dass der Algorithmus einen Zähler auf jedem einzelnen Knoten erstellt. Den ersten Knoten werten alle Threads in jedem Fall aus. Bei allen anderen Knoten wird jedoch nur dann ein Knoten durch einen Thread evaluiert und traversiert, wenn der jeweilige Zähler 0 ist.

Sobald ein Thread einen Knoten betritt, wird der Zähler inkrementiert, bei dem Verlassen des Knotens dekrementiert. Der Zähler wird meist als zusätzliches Attribut in den Transposition-Tables festgelegt.

Eine Studie von Mark Brockington 1994 ergab, dass die Nutzung des ABDADA Konzepts in einem einfachen NegaScout-Algorithmus einen 30%igen Geschwindigkeitsvorteil ergab im Bezug zu dem gleichen NegaScout Algorithmus ohne Parallelisierung [2] [23].

4.6.4 PVS - Principal Variation Splitting

Auch bei PVS starten alle Prozesse mit dem Wurzelknoten. Jeder Thread geht den Baum bis einen Knoten vor dem tiefsten Punkt runter, zur Rekursionstiefe 1. Der erste Thread evaluiert dann den ersten Knoten. Während dieser Zeit warten alle weiteren Prozesse. Sobald der erste Prozess den ersten Knoten evaluiert und einen anfänglichen Alpha-Wert geschaffen hat, werden alle Threads aktiv, und die restlichen Knoten werden parallel ausgewertet. Wenn bis zu diesem Zeitpunkt nur ein einziger Thread existiert für die Auswertung der Principal Variation, kann jetzt entweder für jeden weiteren Teilbaum ein eigener Thread gestartet werden, oder eine feste Anzahl an Threads wird gestartet. Bei einer festen Anzahl an Threads müssen die zu evaluierenden Teilbäume jedoch in einem Stack verwaltet werden, dass Threads ohne Arbeit sich von diesem Stack einen weiteren Teilbaum nehmen können.

Wenn bei der Auswertung der Knoten eine neue Erkenntnis in Form von Alpha- oder Beta-Werten gewonnen wird, wird diese den anderen Threads mitgeteilt. Sind alle Knoten vollständig evaluiert worden, warten die fertigen Threads auf die restlichen oder beenden sich. Wenn alle Threads ihre Suche beendet haben, folgt der rekursive Aufstieg zum nächst übergeordneten Knoten. Da der erste Knoten bereits vollständig evaluiert wurde, kann die Evaluierung der restlichen Knoten unverzüglich parallel durch Threads erfolgen, bis der Wurzelknoten erreicht wird und das endgültige Ergebnis vorliegt.

Eine Problematik des PVS ist jedoch, dass ein Splitting ausschließlich an den Knoten der Principal Variation geschieht. Bei einer Rekursionstiefe von $d = 10$, wird also nur 10 mal gesplittet und nicht, wie beispielsweise bei YBWC an jedem Knoten. Bei einem 4-Kern Prozessor konnte eine durchschnittliche Beschleunigung von 3.0 gemessen werden, mit approximierten Werten von maximal 5.0 [18].

5 Branching-Faktor

Der Branching-Faktor b ist eine meist als arithmetisches Mittel interpretierbare Zahl, die angibt, wieviele mögliche Züge im Durschnitt von einer festen Position aus getätigt werden können. Bei Arimaa ist der Branching-Faktor die Menge aller Spielfeldsituationen, die sich nach den vier einzelnen Bewegungen insgesamt ergibt.

Untersuchungen über alle online gespielten Arimaa-Spiele bis zum Jahre 2005 haben ergeben, dass Arimaa einen durchschnittlichen Branching-Faktor von 17.281 aufweist [19]. Heruntergerechnet auf die einzelnen Ebenen des Suchbaumes ergibt das einen Durchschnitt von 11,47 Möglichkeiten einen Schritt

zu tätigen. Auf jeder Ebene i des Baumes hat der Baum also b^i Knoten. Bei einer Rekursionstiefe d kommt man dabei auf

$$\sum_{i=0}^{d} b^i = 1 + b^1 + b^2 + b^3 + \cdots + b^d$$

Knoten [20]. Auf diese Weise lassen sich alle möglichen Spielkonstellationen für eine beliebige Rekursionstiefe d grob überschlagen, in dem man

$$\mathsf{Anzahl\_Spielkonstellationen} = (\mathsf{Branching\_Faktor})^{\mathsf{Rekursionstiefe}}$$

rechnet.

Das bedeutet, dass alleine der eigene Halbzug mit vier Schritten zu $11,47^4 \approx 17.281$ Spielkonstellationen führt. Bei Berücksichtigung möglicher Schritte des Gegners ergibt das $11,47^8 \approx 299.576.225$ Spielkonstellationen.

Da bei Arimaa eine bestimmte Spielposition mit unterschiedlichen Schritt-Reihenfolgen aus erreicht werden kann, reduzieren sich die genannten Summen etwas. Diese können bei der Traversierung und Evaluierung des Suchbaumes nur von fortgeschritteneren Algorithmen mit Memory zur Laufzeit erkannt und berücksichtigt werden. Einfache Algorithmen, wie der Minimax-Algorithmus (siehe Kapitel 3.1 „Minimax“) ohne Memory-Funktion sind dazu nicht in der Lage und werten konsequent alle Knoten aus.

6 Principal Variation

Die Principal Variation ist eine Folge von Schritten, die als bestmögliche Abfolge dieser gilt, also die Schritte, die zu dem besten Ergebnis führen und somit in der Regel gespielt werden. Eine Principal Variation wäre also in folgender Grafik der rot gekennzeichnete Pfad:

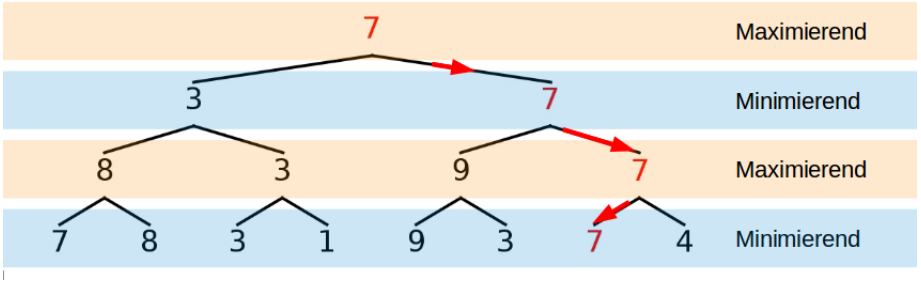


Abbildung 22: *Principal Variation*

Es gibt mehrere Möglichkeiten zur Ermittelung einer Principal Variation. So lässt sie sich bei einer Nutzung von Transposition Tables (siehe 4.3 „Transposition Table“) aus diesen ableiten oder bei einer Nutzung eines Depth-First-Grundalgorithmus beim rekursiven Aufstieg auch anstatt eines einfachen Minimax-Wertes zurückgeben. Je nach Typ des Algorithmus' gibt es verschiedene Möglichkeiten, auf die Principal Variation zuzugreifen, die jeweils kleinere Vor- und Nachteile bieten.

7 Evaluierung

Die Evaluierung einer Spielfeldsituation bildet einen Kernaspekt bei der Entwicklung eines Arimaa-Bots. Hierbei handelt es sich um eine Zustandsbewertung einer Spielfeldsituation. Die Grundalgorithmen, wie Minimax oder MTD(f), traversieren den Zustandsbaum bis zu einer gegebenen Tiefe, oder bis die Zeit abgelaufen ist. Die finalen Knoten oder Blätter des Baumes müssen nun bewertet werden. Die Bewertung muss immer eine nicht abstrakte numerische Zahl liefern, anhand derer verschiedene Zustände des Spielfelds vergleichbar gemacht werden können.

Eine Evaluierung einer Spielfeldsituation kostet Zeit. Je mehr Aspekte bei der Bewertung berücksichtigt werden, je feiner bestimmte strategische Aspekte und Spielphasen integriert und bewertet werden, desto zeitintensiver ist die Evaluierung. Weiterführend bedeutet das, je besser und detailreicher eine Evaluierung ist, desto weniger Knoten des Baumes können in der gleichen Zeit evaluiert werden. Somit verringert sich die mögliche Rekursionstiefe, je komplexer die Evaluierung ist.

Einerseits erlaubt eine extrem komplexe Evaluierung eine sehr gute Einschätzung der aktuellen Situation, andererseits ist man mit einer sehr einfachen Evaluierung in der Lage, unter Umständen so viel tiefer in den Zustandsbaum vorzudringen, dass die einfache Evaluierung in der gleichen Zeit unter Umständen die komplexe übertreffen kann.

Daraus ergibt sich ein genereller Konflikt zwischen Komplexität, Erkenntnisstand und Geschwindigkeit. Welche Form der Evaluierung tatsächlich gewählt wird, liegt im Auge des Betrachters und ist von Programm zu Programm unterschiedlich.

Ein Vorteil einer eher leichtgewichtigeren Evaluierung ist jedoch die Wartbarkeit und Fehlerfreiheit des Programms. In der Praxis ergibt sich daraus ein nicht zu unterschätzender Vorteil. Folgende Grundsätze bilden eine Grundlage für die Entwicklung einer Evaluierungsfunktion [38].

- ***Orthogonalität:*** *Wenn möglich, ist zu vermeiden, dass verschiedene Bestandteile der Evaluierungsfunktion zu einem gewissen Teil gleiche Aspekte evaluieren. Bei der Erweiterung einer Evaluierungsfunktion durch eine neue Komponente muss diese mit allen vorhandenen abgeglichen werden, um bei Überschneidungen die Komponenten zusammenzufassen, sinnvoll zu trennen oder unter Umständen sogar wegzulassen.*

- ***Kontinuität:*** *Wenn zwei verschiedene, gleich gute Spielzustände mittels eines oder einiger weniger guter Halbzüge voneinander erreicht werden können, so sollten beide Spielfeldsituationen ähnlich bewertet werden. Genauso wie die Zustände, welche zwischen beiden Positionen liegen. So sollten, wenn für einen Position ein großer Bonus oder eine große Bestrafung vergeben wird, die Positionen, welche dorthin führen oder welche ähnlich sind, mit einem kleineren Bonus oder einer kleineren Bestrafung versehen werden.*

- ***Progressivität:*** *Es ist sehr viel wichtiger, dass eine Evaluierungsfunktion zwei nahezu identische oder sehr ähnliche Situationen voneinan-*

der unterscheiden kann, anstatt zwei grundsätzlich verschiedene Positionen zu unterscheiden. So ist es beispielsweise weniger wichtig, ein Goal-Threat von einer Middle-Game Situation zu unterscheiden, anstatt den Unterschied zu erkennen, ob ein Hase sinnvoller einen Schritt nach vorne oder zur Seite gehen sollte.

- ***Gutes Worst-Case Verhalten:*** *Ein Bot spielt durchschnittlich deutlich besser, wenn er sich die gesamte Zeit um den Wert eines Hasens verrechnet, als wenn er sich in einem Prozent aller Fälle um den Wert eines Kamels verrechnet. Genau, wie ein guter Arimaa-Spieler nur so gut ist, wie seine schlechtesten Züge und nicht, wie seine besten.*

7.1 Auswirkung der Rekursionstiefe auf die Evaluierung

Viele Prinzipien der Evaluierung können von anderen Spielen, wie Schach oder Go übernommen werden. Es gibt jedoch einen grundlegenden Unterschied, der Arimaa deutlich von den anderen Spielen unterscheidet, insbesondere auch im Bezug auf die Implementierung der Evaluierungsfunktion.

Im Gegensatz zu anderen Brettspielen setzt sich ein Halbzug in Arimaa aus maximal vier unterschiedlichen einzelnen Bewegungen zusammen. Bei der Traversierung des Baumes bildet jede Bewegung eine neue Rekursionstiefe. Doch für die Evaluierung ist es entscheidend zu wissen, wo man sich genau im Zustandsbaum befindet.

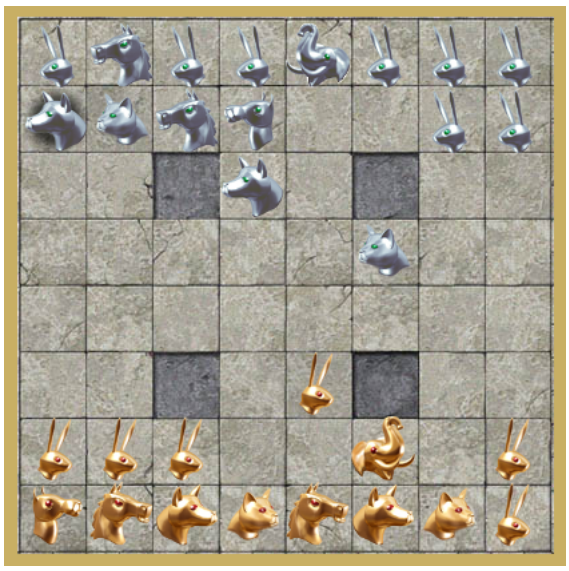


Abbildung 23: *Evaluierung einer Gefährdung*

Betrachtet man beispielsweise die obige Spielfeldsituation aus Sicht von Silber bei einer Rekursionstiefe von 4, gibt es keinerlei Gegner, die näher als 3 Schritte von der silbernen Katze auf f5 positioniert sind. Trotzdem besteht die Gefahr, dass Gold die Katze in seinem Halbzug gefangen nimmt.

Um der Bedrohung einer möglichen Gefangennahme im Halbzug des Gegners vorzubeugen, ist es möglich, eine komplexe Evaluierungsfunktion zu schreiben, welche in der Lage ist, alle Bedrohungen eigener Figuren zu erkennen,

selbst wenn dabei alle vier Schritte des Gegners benötigt werden. Dies geschieht mittels eines aufwändigen Entscheidungsbaumes.

In einem anderen Fall aber wird ein Spielfeldzustand nach 6 Rekursionstiefen evaluiert. Die genannte Evaluierungsfunktion prüft jetzt den Zustand und findet einige Bedrohungen und versucht, diese zu sichern. Allerdings ist ein Großteil davon keine tatsächliche Gefahr, denn Gold hat nur noch zwei Schritte übrig und nicht, wie in der Evaluierungsfunktion angenommen, vier.

Dies führt dazu, dass eine Evaluierungsfunktion bei einer höheren Rekursionstiefe dafür sorgt, dass der Bot extrem vorsichtig spielt, da vielfach eine Gefahr angenommen wird, die noch gar nicht konkret vorhanden ist. Anders herum jedoch würde eine Evaluierungsfunktion, die für eine Rekursionstiefe von 7 ausgelegt ist, bei einer konkreten Rekursionstiefe von 5 gar keine Gefahr erkennen, obwohl unter Umständen Figuren komplett entblößt sein könnten.

Zusammenfassend bedeutet das, dass man bei Arimaa entweder für jede mögliche Rekursionstiefe (Also 0-3) eine separate Evaluierungsfunktion schreibt, die den Zustand entsprechend korrekt bewertet, oder sich bei der gewählten Rekursionstiefe auf eine feste Rekursionstiefe festlegt, welche auf die Evaluierungsfunktion passt und in der Zeit für einen Zug problemlos zu bewältigen ist.

Denn es ist verheerend, wenn aus Zeitnot einige Rekursionstiefen nicht traversiert werden können, wodurch die Evaluierung vorhandene kritische Situationen nicht erkennt. Durch eine Gewichtung der Evaluierung abhängig von der jeweiligen Rekursionstiefe wird eine Nutzung einer Evaluierungsfunktion auch für unterschiedliche Rekursionstiefen möglich.

Hierbei wird der evaluierten Wert der eigenen Position und die des Gegners gewichtet. Besteht bei einer Spielfeldsituation eine Gefahr für beide Spieler, wird diese für den Spieler niedriger gewichtet, der am Zug ist, da er auf die Gefahr noch reagieren kann. Je mehr Schritte verfügbar sind, desto geringer fällt die Gewichtung aus. Diese Form der Gewichtung ermöglicht es, dass eine Gefahr, welche unter Umständen gar nicht vorhanden ist, geringer gewichtet wird. Das verhindert aber nicht gänzlich, dass die Situation prinzipiell falsch evaluiert wird.

7.2 Statisch

Eine statische Evaluierung bewertet einen Knoten eines Zustandsbaumes, meist ohne Informationen über den Werdegang und die Hinführung zu der Situation zu sammeln und zu bewerten. Die Klassifizierung in eine statische und dynamische Evaluierung ist eine eigenständige und nicht auf bestehenden Arbeiten resultierende Aufteilung der Evaluierung um die Unterschiede beider zu verdeutlichen und herauszustellen.

7.2.1 Materialevaluierung

Bei der Materialevaluation wird bewertet, wie stark ein Spieler im Verhältnis zu seinem Gegner darsteht, ausschließlich beruhend auf der Menge und Art der Figuren, die sich auf dem Spielfeld befinden. Wie in anderen Spielen ist eine Materialbewertung essentiell für das Spiel. Eine Schwierigkeit ist jedoch, dass in Arimaa Material dynamisch bewertet werden sollte. Ein Hund kann

fast den gleichen Wert haben, wie ein Pferd, sobald der Gegner selbst keine Hunde mehr besitzt. Genauso führt der Verlust eines Kamels dazu, dass die Pferde des anderen Spielers höher bewertet werden sollten.

7.2.1.1 Einfache Materialevaluierung

Bei der einfachen Evaluierung des Materials wird jedem Figurtyp ein Multiplikator zugewiesen, welcher den Figuren eine Gewichtung zueinander ermöglicht. Die Gewichtungen der einzelnen Figurtypen erfolgt aus Erfahrungen und Tests bereits veröffentlichter Arbeiten zum Thema Arimaa [5] und ist wie folgt:

		16
		11
		7
		4
		2
		1

Tabelle 5: *Gewichtung der einzelnen Figurtypen*

Alle vorhandenen Figuren werden mit ihrem Multiplikator multipliziert und addiert. Danach wird der Gesamt-Materialwert von Silber dem von Gold abgezogen, um beide Spieler in einen Zusammenhang zu stellen. Der dynamische Aspekt von sich verändernden Materialwerten im Verhältnis zum Gegner ist hier außer Acht gelassen.

Weiterhin gibt es eine Bestrafung, wenn sich weniger als vier Hasen des jeweiligen Spielers auf dem Spielfeld befinden. Je weniger Hasen, desto höher die Bestrafung. Dies dient dem Zweck, ein vorschnelles Opfern von Hasen im Endgame zu verhindern und eine prinzipielle Ziel-Bedrohung aufrecht zu erhalten.

7.2.1.2 HarLog-Evaluierung

Die Harlog-Evaluierung ist eine heuristische Evaluierung, welche auch von hochrangigen Bots, wie Bot_Sharp eingesetzt wird. Die Werte der HarLog-Funktion sind nicht wissenschaftlich belegt und sind ausgewählt und getestet mit dem Ziel Materialwerte im Sinne des jeweiligen Entwicklers zu erzeugen [54]. Die zentrale Idee der HarLog-Funktion ist es, variable Materialwerte zu erzeugen, abhängig von der Anzahl stärkerer Figuren auf dem Board.

Die Berechnung des Materialwertes geschieht auf folgende Weise [54]:

- Setzen der Variablen $Q = 1.447530126$ und $G = 0.6314442034$.
- Für jede Figur der eigenen Farbe, außer des Hasens, mit keiner stärkeren gegnerischen Figur, addiere: $2 / Q$.
- Für jede Figur der eigenen Farbe, außer des Hasens, mit n stärkeren gegnerischen Figuren, addiere: $7 / (Q + n)$.
- Setzen der Variablen r auf die Anzahl von Hasen der eigenen Farbe und t auf die Gesamtzahl der Figuren der eigenen Farbe.
- Addiere: $G(log(r) + log(t))$.

7.2.1.3 HERD - Holistic Evaluator of Remaining Duels

Der HERD-Evaluator ist eine Bewertung von Materialwerten, welche die Stärke von Figuren und daraus ein gesamtheitliches Verhalten berechnet. Die zentrale Idee von HERD ist es, einen eigenständigen Material-Evaluator zu erzeugen, welcher nahezu ohne jegliche Parameter und per Hand einzustellende Variablen auskommt. Lediglich die Anzahl an Figuren ist relevant und nicht konstant.

HERD unterscheidet dabei auch zwischen den einzelnen Vorkommen und Anzahl von Figuren des gleichen Typs um eine variable Stärke von Figuren zu erzeugen, abhängig ihrer Anzahl auf dem Spielfeld. Für eine kompaktere Schreibweise sind die Figuren auf folgende Weise abgekürzt.

goldener Elefant	E
goldenes Kamel	M
goldenes Pferd	H
goldener Hund	D
goldene Katze	C
goldener Hase	R

Tabelle 6: *Abkürzungen für Figurtypen*

Für Silberne Figuren gelten die gleichen Abkürzungen in Form von Kleinbuchstaben. Eine Hierarchie der einzelnen Figuren geschieht auf folgende Weise:

$$E > M > H > D > C > R > \textit{gefangen genommene Figuren}$$

Dabei wird zusätzlich unterschieden zwischen dem ersten und letzten Vorkommen einer Figur. Eine Letzte-Figur ist immer stärker oder mehr wert, als eine Erst-Figur. Der Effekt ist, dass ein Opfern eines zweiten Pferdes verhindert

wird und somit eine Lücke zwischen dem Kamel und den Hunden geschaffen wird, welche die gegnerischen Pferde stärken würde.

Bei Hasen jedoch ist die Reihenfolge umgedreht. Hier ist die Erst-Figur die wertvollste. Der Sinn dahinter ist es, ein Opfern von Hasen generell zu verhindern oder aufzuschieben, um im späteren Spielverlauf genug Figuren zum Halten von Traps und für Ziel-Bedrohungen zu haben.

Eine grundsätzliche vollständige Hierarchie ist also folgende:

$$E > M > \textit{Last } H > \textit{1st } H > \textit{Last } D > \textit{1st } D > \textit{Last } C > \textit{1st } C > \textit{1st } R > \\ \ldots > \textit{8th } R > \textit{gefangen genommene Figuren}$$

Die gesamte HERD-Evaluierung beruht auf folgender mathematischen Definition. Für die folgenden Formeln sei aufgrund der kompakteren und übersichtlicheren Schreibweise folgendes definiert:

$$\begin{aligned} N_i &\mathrel{\widehat{=}} Gold\_Type\_Piece_i \\ n_i &\mathrel{\widehat{=}} Silver\_Type\_Piece_i \\ G &\mathrel{\widehat{=}} Gold \\ S &\mathrel{\widehat{=}} Silver \end{aligned}$$

Die Indizes i, j und k der folgenden Formeln, welche auf goldene und silberne Figuren bezogen sind, referenzieren die folgende Tabelle:

1	Elefant
2	Kamel
3	Letztes Pferd
4	Erstes Pferd
5	Letzter Hund
6	Erster Hund
7	Letzte Katze
8	Erste Katze
9	Erster Hase
10	Zweiter Hase
11 . . . 15	. . .
16	Letzter Hase
17	Gefangen genommene hohe Figuren[1]
18	Gefangen genommene Hasen
19	Auf dem Spielfeld verbliebende Hohe Figuren
20	Auf dem Spielfeld verbliebende Hasen

Tabelle 7: *Bedeutung der in den Formeln verwendeten Indizes*

$$\mathsf{Strength_i}(\mathsf{N_i}) = \mathsf{N_i} \sum_{\mathsf{j=i}}^{18} \mathsf{n_j}$$

Die *Strength()*-Funktion multipliziert die Anzahl der vorhandenen Figuren von sich selber mal der Summe aller gegnerischen Figuren, die schlechter oder gleich gut sind.

$$\mathsf{Power}(\mathsf{N_i}) = \sum_{\mathsf{j=1}}^{\mathsf{i}} \mathsf{Strength}(\mathsf{N_j})$$

Die *Power()*-Funktion ist die Summe der *Strength()* aller besseren eigenen

[1] Hohe Figuren sind alle Figuren außer Hasen

Figuren. Durch *Power()* wird also die Unterstützung durch bessere eigene Figuren ausgedrückt, wobei höhere Figuren weniger Unterstützung haben, aber mehr *strength()*.

$$\mathsf{Balance}(\mathsf{N_i})) = \frac{\mathsf{Power}(\mathsf{N_i})}{\mathsf{Power}(\mathsf{N_i}) + \mathsf{Power}(\mathsf{n_i})}$$

Balance() ermittelt aus den konkreten Zahlen der *Power()* ein Verhältnis, wie sich die *Power()* bezüglich einer bestimmten goldenen Figur verhält. Die Zahl ist prozentual und liegt zwischen *0..1*.

$$\mathsf{Bias}(\mathsf{G}) = \frac{(\mathsf{N_{18}} + \mathsf{n_{18}}) * \mathsf{N_{20}}}{\mathsf{n_{19}}}$$

Bias() ermittelt eine Ausrichtung oder Tendenz einer möglichen Gewinnchance durch das Inbetrachtziehen gefangengenommener Hasen, auf dem Brett verbliebene Hasen und verbliebener hoher Figuren.

$$\mathsf{Balance\_Goals}(\mathsf{G}, \mathsf{Goals}) = \frac{\mathsf{Bias}(\mathsf{G})}{\mathsf{Bias}(\mathsf{G}) + \mathsf{Bias}(\mathsf{S})}$$

Balance() ergibt eine Wahrscheinlichkeit bezüglich der Gewinnchance von entweder Gold- oder Silber.

$$\mathsf{Herd}(\mathsf{G}) = \frac{\sum_{\mathsf{i}=1}^{16}(\mathsf{N_i} + \mathsf{n_i}) * \mathsf{Balance}(\mathsf{N_i}) + \mathsf{Bias}(\mathsf{G})}{\sum_{\mathsf{i}=1}^{16}(\mathsf{N_i} + \mathsf{n_i}) + \mathsf{Bias}(\mathsf{G}) + \mathsf{Bias}(\mathsf{S})}$$

Daraus setzt sich dann eine Gesamtformel zur Berechnung des globalen Materialwertes auf dem Spielfeld (Immer gerechnet für Gold) zu folgender Formel zusammen:

$$\begin{aligned}\mathsf{Herd}(\mathsf{G}) = & \frac{1}{\sum_{\mathsf{i}=1}^{16}(\mathsf{N_i} + \mathsf{n_i}) + \frac{(\mathsf{N_{18}} + \mathsf{n_{18}}) * (\mathsf{N_{19}}\mathsf{N_{20}} + \mathsf{n_{19}}\mathsf{n_{20}})}{\mathsf{N_{19}}\mathsf{n_{19}}}} \\ & * (\sum_{\mathsf{i}=1}^{16}(\mathsf{N_i} + \mathsf{n_i}) * \frac{\sum_{\mathsf{j}=1}^{\mathsf{i}} \mathsf{N_j} \sum_{\mathsf{k}=\mathsf{j}}^{18} \mathsf{n_k}}{\sum_{\mathsf{j}=1}^{\mathsf{i}} \mathsf{N_j} \sum_{\mathsf{k}=\mathsf{j}}^{18} \mathsf{n_k} + \sum_{\mathsf{j}=1}^{\mathsf{i}} \mathsf{n_j} \sum_{\mathsf{k}=\mathsf{j}}^{18} \mathsf{N_k}} \\ & + \frac{(\mathsf{N_{18}} + \mathsf{n_{18}}) * \mathsf{N_{20}}}{\mathsf{n_{19}}})\end{aligned}$$

Alle Berechnungen beziehen sich ausschließlich auf die Berechnung für den goldenen Spieler. Da die Berechnung jedoch eine Verhältniszahl ermittelt, lässt sich das Ergebnis für Silber ableiten, in dem nach der vollständigen Berechnung das Ergebnis mit -1 multipliziert wird. Also:

$$\mathsf{Herd(S)} = \mathsf{Herd(G)} * -1$$

7.2.2 Goal-Detection

Die Goal-Detection beinhaltet eine Überprüfung, ob das Spiel gewonnen oder verloren ist und bildet daher einen extrem wichtigen Bestandteil jeder Evaluierung. Das Ergebnis dieses Teils der Evaluation erzeugt drei verschiedene Werte. Gewonnen, Verloren oder 0, wenn keines der beiden Ereignisse eingetreten ist.

Dabei werden die zwei Hauptaspekte überprüft:

- Hat ein eigener oder gegnerischer Hase die jeweilige Ziellinie erreicht?
- Hat der Gegner oder die eigene Farbe keinen Hasen mehr?

7.2.3 Trapkontrolle

Die Trapkontrolle bildet den umfangreichsten und aufwändigsten Teil der Evaluierung. Dabei muss jeder der vier Traps bezüglich verschiedener Aspekte betrachtet und bewertet werden . Die Auswirkung der Rekursionstiefe auf die Evaluierung hat seine größte Auswirkung auf die Trapkontrolle.

Die behandelten Aspekte der Trapkontrolle sind:

- Gefangennahme von Figuren
- Sicherung eigener Figuren und zerstören von Sicherungen gegnerischer Figuren
- Gefahrerkennung bei der Nähe zu Traps
- Gefahrerkennung durch höhere gegnerische Figuren
- Erkennung von False Protection
- Bewertung der Situation von Traps und die Zugehörigkeit zu einem Spieler
- Traps gegen gegnerische Übernahmen sichern
- Freezing in Trapnähe

Da es sich bei der Sicherung und Kontrolle von Traps um einen Schlüsselaspekt der gesamten Kontrolle des Spielfelds und Positionierung handelt, ist es hier besonders wichtig, dass die Trapkontrolle für genau die Rekursionstiefe implementiert wurde, wie sie nachher aufgerufen wird. Je genauer und besser die Trapkontrolle ist, desto weiter kann man prinzipiell Schritte des Gegners vorausschauen, zusätzlich zu der erreichten Rekursionstiefe.

Um die Trapkontrolle möglichst effizient und übersichtlich zu gestalten, ist die Bildung eines Entscheidungsbaumes eine meist gute Herangehensweise. Für eine Evaluierung für eine Rekursionstiefe von d = 6 und zur Überprüfung der Gefangennahme von Figuren hat ein Entscheidungsbaum in etwa folgende Ausprägung:

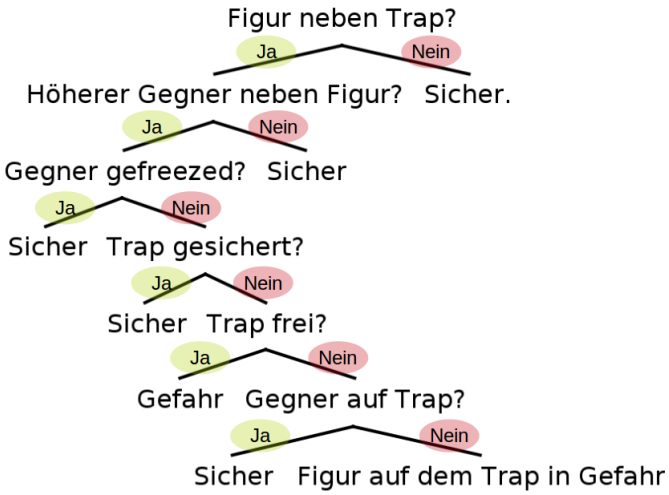


Abbildung 24: *Vereinfachter Entscheidungsbaum einer Gefangennahme für d = 6*

In der aufgezeigten Abbildung werden innerhalb der Evaluierung der speziellen Situation einer Gefangennahme die zwei verbleibenden Schritte des Gegners mitberechnet, was die einer endgültigen Evaluierung einer Rekursionstiefe von $d = 8$ entspricht und somit einen entscheidenden Teil der Traversierung des Suchbaumes übernimmt.

In der gleichen Form ist es möglich, einen Entscheidungsbaum zu implementieren, der drei oder sogar vier Schritte des Gegners voraussehen und bewerten kann. Allerdings wird schnell deutlich, dass die Anzahl der Möglichkeiten eines Captures in unserem Fall exponentiell steigt, je mehr Schritte des Gegners in dem Entscheidungsbaum berücksichtigt werden. Besonders, da auch weiter Umstände wie Blockaden, Freezing, weitere Gegner oder False Protection hinzu kommen können.

Die Gefahrerkennung, Sicherung eigener Figuren und das Zerstören der Sicherungen gegnerischen Figuren fällt in die grundsätzlich gleiche Kategorie, wie das Gefangennehmen einer gegnerischen Figur. Da die Gefangennahme einer Figur oder die evaluierte Gefahr einer Gefangennahme negativ evaluiert wird, tendiert der Bot dazu, entweder den Trap zu sichern, einen Gegenangriff zu starten oder den angreifenden Gegner außer Gefecht zu setzen.

Die Erkennung einer False-Protection ist hauptsächlich relevant bei einer Evaluierung von vier Schritten durch die Evaluierungsfunktion. Dabei muss erkannt werden, dass ein Trap, welcher durch zwei eigene Figuren unterstützt wird, zur eigenen Falle werden kann, wenn beide Figuren einen höheren Gegner neben sich haben. Diese können die Figuren einerseits in den Trap schieben,

andererseits die zweite unterstützende Figur wegschieben oder ziehen, wodurch die sich im Trap befindliche Figur gefangengenommen wird.

Zur Bewertung der generellen Trapsituation gehört die Beurteilung, welcher Spieler die Kontrolle über den Trap besitzt und in welchem Ausmaß. Also wie sicher der Trap in der eigenen Hand ist. Davon ist abhängig, ob versucht wird, Hasen nach vorne zu ziehen oder die Aufmerksamkeit auf andere Bereiche des Spiels zu lenken.

Immobilisierung gehört zu den Kernaspekten in Trapnähe. Denn Figuren, die sich nicht bewegen können, sind in genereller Gefahr, in naher Zukunft in den Trap gezogen/geschoben zu werden. Weiterhin benötigt der Gegner zusätzliche Schritte, um seine Figuren zu sichern, anstatt einen eigenen Angriff zu starten. Somit bildet das Freezing in Trapnähe einen weiteren Aspekt, der zur allgemeinen Kontrolle des Traps beiträgt.

7.2.4 Movement

Bewegungsfreiheit ist im generellen Kontext von Arimaa wichtig. Zu wenig Bewegungsrichtungen können schnell zu Blockaden der eigenen Figuren führen, welche zu spät registriert wurden. Ähnlich, wie Hostage-Situationen und False-Protection in Trapnähe. Einige der genannten Punkte müssen separat behandelt und untersucht werden. Doch mittels einer Evaluierung der generellen Bewegungsfreiheit einzelner Figuren kann eine Tendenz erreicht werden, die unter Umständen frühzeitig dafür sorgt, dass man nicht in Situationen kommt, in denen eine Hostage-Situation beispielsweise nicht mehr verhinderbar ist.

Je nach Typ der Figur spielt die Bewegungsfreiheit eine größere oder kleinere Rolle. So ist die Bewegungsfreiheit des Elefanten extrem wichtig, während ein bewegungsunfähiger Hase den Spielfluss meist nicht stört. Weiterhin gibt es Strategien, in denen zum Beispiel ein Pferd mit Absicht in eine Hostage-Situation gebracht wird, um den gegnerischen Elefanten zu binden, während man auf der anderen Seite des Spielfelds mit seinem Kamel und dem zweiten Pferd einen Angriff startet.

Eine Bewegungsfreiheit ist also wichtig, aber situationsbedingt unterschiedlich zu beurteilen.

7.2.5 Hostage

Eine Hostage-Situation ist ein Spezialfall, bei dem eine eigene Figur bei einem Trap hinter oder in die gegnerischen Linie gezogen wird. Die Figur ist meist bewegungsunfähig, und es besteht die Bedrohung, dass diese Figur in einem Halbzug gefangengenommen wird. Somit ist der Elefant verpflichtet, sich dauerhaft neben dem Trap zu positionieren, da sonst ein Materialverlust zu erwarten ist.

Diese temporäre Bewegungsunfähigkeit wird meist ausgenutzt, um einen Angriff auf die andere Spielfeldhälfte zu starten. Eine Hostage-Situation kann jedoch auch zu dem eigenen Vorteil gewertet werden, und es muss sehr genau abgewogen werden, ob man das Risiko eingehen möchte. Im Allgemeinen versucht ein Spieler jedoch eine Hostage-Situation eigener Figuren zu verhindern.

Die Evaluierung einer Hostage-Situation ist daher stark von Figuren der näheren Umgebung und deren Stärke abhängig. Zusätzlich ist abzuwägen, ob

die Situation einen Gegenangriff erlaubt und man in der Lage wäre, einen Angriff auf der anderen Seite des Spielfeldes abzuwehren.

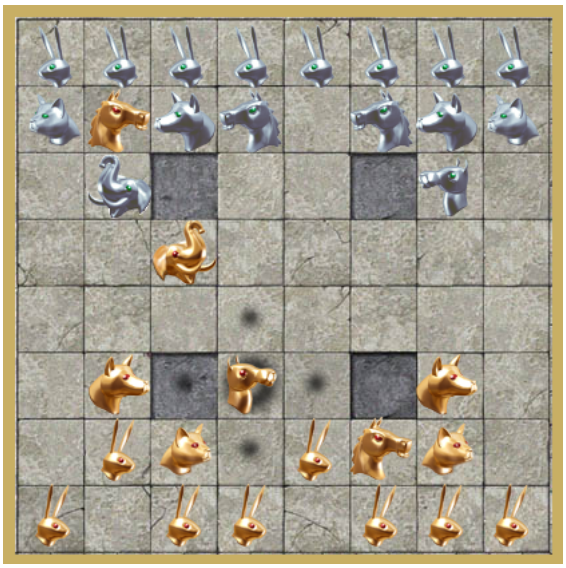


Abbildung 25: *Beispiel einer Hostage-Situation*

7.2.6 Frame

Die Evaluierung einer Frame-Situation wird meist eine höhere Bedeutung zugewiesen, als einer Hostage. Beide Bedrohungen können jedoch ähnlich evaluiert werden. Bei einem Frame steht die bedrohte Figur direkt auf einem Trap und ist von drei Seiten umringt mit Gegnern. Meist wird der Figur keine Möglichkeit gegeben, sich aus dem Trap zu schieben. Die beschützende Figur ist meist der Elefant, um eine vollständige Sicherheit zu gewähleisten. Solange der Frame aufrecht erhalten ist, gibt es keine Möglichkeit, den Elefanten zu bewegen, ohne einen direkten Materialverlust zu erleiden. Der Gegner wiederum kann langsam seine hohen Figuren aus der Frame-Situation heraus rotieren und ist dann meist im Besitz des Strongest Free Piece auf dem Rest des Spielfeldes. Dies erlaubt es meist einen Materialvorteil zu gewinnen, bis der Frame aufgebrochen oder aufgelöst wird.

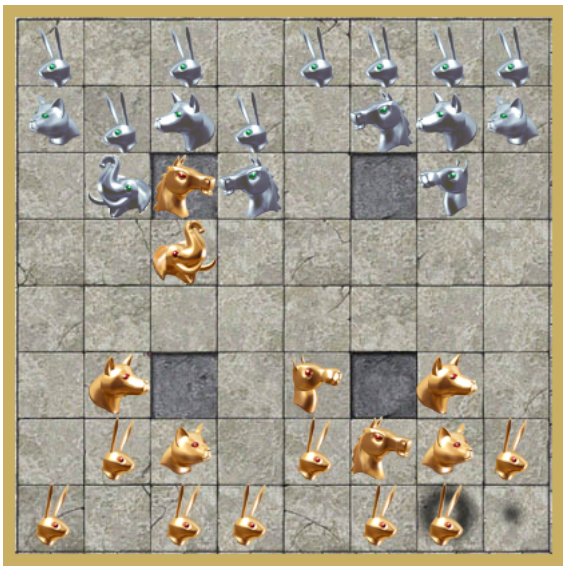


Abbildung 26: *Beispiel einer Frame-Situation*

7.2.7 Fork

Eine Fork-Situation entspricht der Bedrohung einer Gefangennahme einer Figur in zwei verschiedenen Traps innerhalb des folgenden Halbzuges des Gegners. Sie bildet daher eine große Gefahr für den Spieler, dessen Figur der Bedrohung ausgesetzt ist. Eine Fork-Situation bedingt, dass die bedrohte Figur sich auf einem der beiden Felder zwischen zwei nebeneinander liegenden Traps befindet. Meist handelt es sich um die Traps der 3. oder 6. Reihe, da es gerade zu Beginn eines Spiels schwer ist, den Trap eines Gegners vollständig zu kontrollieren.

Die bedrohte Figur muss in dem Moment gefreezed sein, und beide der möglichen Traps dürfen keine Unterstützung durch Figuren der gleichen Farbe wie die der bedrohten Figur aufweisen. Wenn die genannten Kriterien erfüllt sind, fällt es dem bedrohten Spieler meist schwer, eine Gefangennahme der Figur in einer der beiden Traps zu verhindern. Meist reichen die verfügbaren Schritte gerade aus, um einen der beiden Traps zu sichern. Oder die sichernden Figuren bringen sich in der Nähe des gegnerischen Traps in Gefahr, selber gefangengenommen zu werden.

Die Evaluierung einer solchen Situation ist entscheidend, um einen Materialvorteil herbeizuführen oder im eigenen Fall frühzeitig zu verhindern. Das Erkennen einer Fork-Situation hingegen ist nicht sehr komplex, da nur wenige spezifische Kriterien erfüllt werden müssen.

Bei der Bedrohung gegnerischer Figuren durch eine Fork-Situation muss beachtet werden, dass alle eigenen Figuren in absoluter Sicherheit sind, bevor die Fork-Situation in einem Austausch von Figuren oder eine strategisch ungünstige Situation umschlägt.

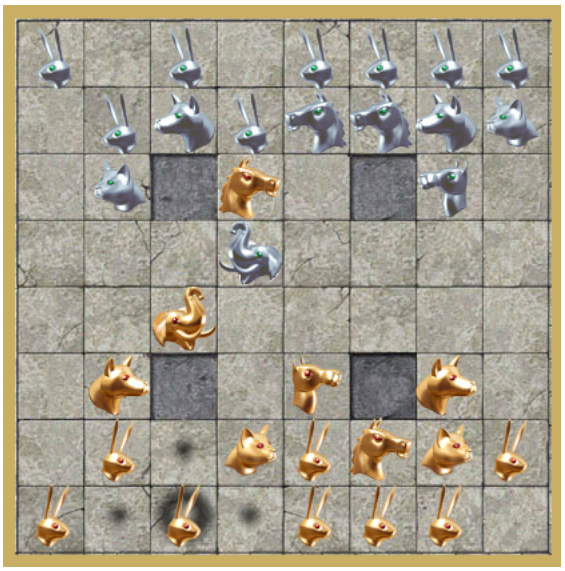


Abbildung 27: *Beispiel einer Fork-Situation*

7.2.8 Piece-Square-Tables

Piece-Square-Tables bieten eine sehr schnelle und einfache Möglichkeit, eine grobe Form der Positionierung von Figuren und Figurtypen auf dem Spielfeld vorzunehmen. Festgelegt ist dafür eine Tabelle der Größe des Spielfeldes. Für jedes Feld ist ein Wert festgelegt. Bei einer Positionierung auf dem Spielfeld wird jeder Figur der Wert des Feldes, auf dem er steht, aus seiner entsprechenden Tabelle geladen und mit der restlichen Evaluierung addiert.

Auf diese Weise bekommen Figuren die Tendenz, auf höherwertigen Feldern zu stehen und somit strategisch eine bessere Position einzunehmen, welche über die Piece-Square-Tables definiert wurde.

7.2.8.1 Positionierung der Figuren

Die Positionierung der Figuren auf dem Spielfeld spielt eine Schlüsselrolle im Spiel. Unterschiedliche Figuren müssen unterschiedliche Bewegungsrichtlinien verfolgen. Dafür werden die Figuren in verschiedene übergeordnete Ränge eingeordnet. HIGH, MIDDLE und LOW. Die Zuordnung geschieht auf folgende Weise:

HIGH	Elefant, Kamel, Pferd
MIDDLE	Hund, Katze
LOW	Hase

Tabelle 8: *Zuordnung von Piece-Square-Tables zu Figurtypen*

Für hohe Figuren ist es besonders sinnvoll, sich tendenziell mittig und zentral zu positionieren, um schnell kritische Spielfeldbereiche zu erreichen und anzugreifen oder zu verteidigen. Das folgende Piece-Square-Table dient der Positionierung hoher Figuren:

HIGH:

8	0	0	1	2	2	1	0	0
7	0	2	2	3	3	2	2	0
6	1	2	-1	4	4	-1	2	1
5	2	3	4	5	5	4	3	2
4	2	3	4	5	5	4	3	2
3	1	2	-1	4	4	-1	2	1
2	0	2	2	3	3	2	2	0
1	0	0	1	2	2	1	0	0
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Tabelle 9: *Piece-Square-Table: HIGH*

Bei mittleren Figuren, wie Hund und Katze, ist eine mittige Positionierung eher ungünstig. Sie sind vor allem in den eigenen Reihen gut positioniert und zur Verteidigung und Trapkontrolle der Home-Traps. Daher ergibt sich eine folgende Tabelle für mittlere Figuren:

MIDDLE:

8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	0	0	-1	0	0	-1	0	0
5	0	0	1	0	0	1	0	0
4	1	1	2	1	1	2	1	1
3	3	6	-1	5	5	-1	6	3
2	2	4	6	4	4	6	4	2
1	1	2	3	2	2	3	2	1
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Tabelle 10: *Piece-Square-Table: MIDDLE*

Die Hasen hingegen sollen die Mitte komplett meiden und eher versuchen, über die Ränder in die gegnerischen Linien zu kommen, um bei Gelegenheit die Ziellinie zu erreichen. Das folgende Piece-Square-Table resultiert daraus:

LOW:

8	8	8	8	8	8	8	8	8
7	7	6	5	4	4	5	6	7
6	6	2	-1	0	0	-1	2	6
5	5	1	0	0	0	0	1	5
4	4	1	0	0	0	0	1	4
3	3	2	-1	0	0	-1	2	3
2	2	2	2	1	1	2	2	2
1	2	2	2	2	2	2	2	2
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Tabelle 11: *Piece-Square-Table: LOW*

7.2.8.2 Defensives und Offensives Spiel

Genau die gleiche Form von Piece-Square-Tables kann auch eingesetzt werden, um die Geschwindigkeit und Angriffslaune des Programms zu beeinflussen, indem man die Figuren dazu ermutigt, nach vorne zu ziehen oder eher in den eigenen Reihen zu bleiben. Das folgende Piece-Square-Table wird auf alle Figuren unabhängig ihres Typs angewandt und sorgt dafür, dass ein Angriff langsamer und sicherer vonstatten geht:

SPEED:

8	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1
6	1	1	-1	1	1	-1	1	1
5	2	2	2	2	2	2	2	2
4	2	2	2	2	2	2	2	2
3	3	3	-1	3	3	-1	3	3
2	4	4	4	4	4	4	4	4
1	3	3	3	3	3	3	3	3
	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**

Tabelle 12: *Piece-Square-Table: SPEED*

7.2.9 Positionierung spezieller Figuren

Abgesehen von einer generellen Positionierung der Figuren durch Piece-Square-Tables ist es besonders für bestimmte Figuren sinnvoll, eine individuelle Positionierung vorzunehmen. Ein Beispiel einer solchen Positionierung ist die eines Kamels und eines Elefanten.

Eine, besonders von Bots und vielen Spielern viel genutzte Strategie ist es, durch eine Hostage oder Frame-Situation, das gegnerische Kamel in die eigene Gewalt zu bringen. Dadurch wird sowohl das Kamel gebunden als auch der Elefant des gegnerischen Spielers, der sich voll und ganz der Verteidigung des Kamels widmen muss, um dieses nicht zu verlieren.

Um dies zu verhindern, reicht eine statische Positionierung durch Tabellen nicht aus. Hier muss eine eigene Form der Evaluierung greifen, die eine Bestrafung einführt, wenn das Kamel sich in zu geringer Entfernung des gegnerischen Elefanten befindet. Diese Bestrafung muss gelten, solange der gegnerische Elefant frei bewegbar und nicht blockiert oder an die Verteidigung wichtiger Figuren gebunden ist. Vorsichtsmaßnahmen sind bei der Evaluierung jedoch zu

treffen, dass der gegnerische Elefant nicht eine Hostage-Situation eines Hundes aufgibt, aber dafür das Kamel in seine Gewalt bekommt.

Der Entwurf und die Höhe der genannten Bestrafung ist in bestimmten Situationen kritisch, da das Halten einer stragetisch wichtigen Situation wichtiger sein kann, als der Abstand zum gegnerischen Elefanten. Weiterhin kann ein Elefant (fast) ungefährlich sein, solange dieser an die Verteidigung hoher Figuren gebunden oder blockiert ist.

Die Evaluierung muss alle diese Faktoren einbeziehen, um einerseits einen ausreichenden Abstand des Kamels zum gegnerischen Elefanten in der normalen Spielsituation zu gewährleisten und andererseits eine strategisch bessere Position nicht hierfür aufzugeben.

7.2.10 Verteidigung, Zusammenhalt und Angriffe von Hasen

Im Anfangs- und Mittelspiel spielen Hasen eine eher untergeordnete Rolle. Doch gerade im Mittelspiel ist es bedeutend, dass Hasen eine gute Positionierung einnehmen, um in einem Endspiel die gegnerischen Linien massiv bedrohen zu können.

Abgesehen von einem möglichen Endspiel sind Hasen durch ihre Vielzahl sehr gut dafür geeignet eine starke Verteidigung gegen schnell vorziehende gegnerische Hasen zu bilden. Hierfür ist ein Zusammenhalt und eine gute Positionierung der Hasen und anderer Figuren essentiell.

Bei der Evaluierung der Brettsituation kann eine gute und starke Positionierung von Hasen erreicht werden, indem jeder Hase Bonuspunkte in bestimmten Situationen bekommt, zum Beispiel wenn sich rechts, links, über oder unter der Figur eine Figur des eigenen Teams befindet. Da die letzten Reihen hauptsächlich mit Hasen besetzt sind, wird erreicht, dass die Hasen eine solide Wand bilden, die ein einfaches Durchkommen fremder Hasen erschwert. Da bei der Evaluierung aber auch andere Figuren als Hasen berücksichtigt werden, werden diese bei jeder Gelegenheit mit in die Blockade der letzten Reihen einbezogen.

Durch die genannte Evaluierung ist eine gute Verteidigung durch Hasen erreicht worden. Doch auch ein Angriff und eine Vorwärtsbewegung von Hasen ist wichtig. Dabei kann allgemein festgestellt werden, dass eine freie Spalte vom Hasen zur Ziellinie eine gute Approximierung einer Angriffsformation bildet. Weiterhin ist eine freie Spalte bis zur Ziellinie wichtiger, je näher der Hase dieser kommt.

Um dies bei einer Evaluierung zu gewährleisten, werden Bonuspunkte vergeben für freie Spalten zwischen dem Hasen und der Ziellinie. Je näher der Hase dabei der Ziellinie ist, desto höher ist der Bonus. Der erzielte Effekt dabei ist es, einen Hasen schneller vorzuziehen, je näher er dem Ziel ist, wenn eine freie Spalte vorhanden ist. Der genannte Bonus ermutigt die eigenen hohen Figuren gleichzeitig dazu, durch Ziehen und Schieben freie Spalten zu schaffen.

7.2.11 Strongest Free Piece

Eine der wichtigsten und auch schwierigsten Berechnungen bei einer Evaluierung ist die Berechnung des *Strongest Free Piece*. Dabei muss durch das Programm erkannt werden, welcher Spieler die höchste freie Figur auf einem Teil des Spielfelds besitzt. Die höchste Figur zu erkennen ist dabei trivial. Jedoch ist es vielfach der Fall, dass ein Kamel oder ein Elefant an die Verteidigung bestimmter Figuren gebunden ist. Eine einfache Form der Gebundenheit entsteht durch eine Frame oder Hostage-Situation.

Weiterhin spielt die Stärke der Gebundenheit einer Figur eine entscheidende Rolle. Betrachtet man die folgende Situation:

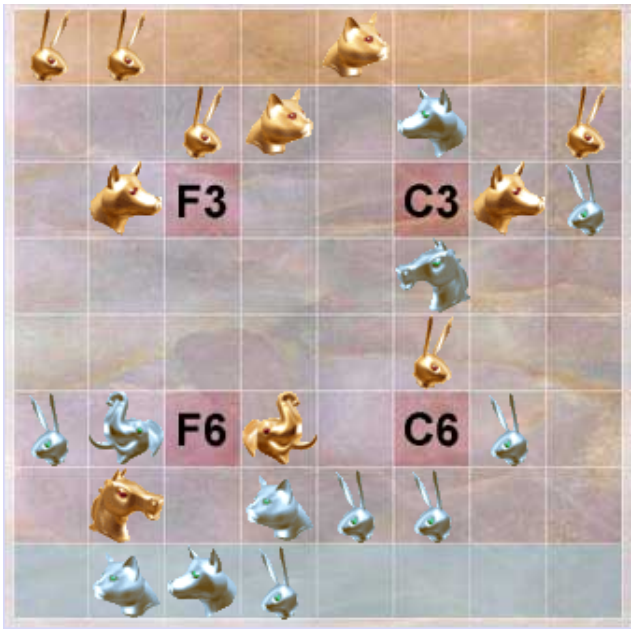


Abbildung 28: *Hostage-Situation und Strongest Free Piece*

Die obige Grafik stellt eine Positionierung im Mittel-/Endspiel eines Spiels mit der Arimaa-Gameroom-Id *290506* und Zug 67g dar. Gold ist also am Zug. Zu erkennen ist eine klassische Situation eines Hostage am f6-Trap. Das goldene Pferd ist bewegungsunfähig und kann, sobald der goldene Elefant seine Position verlässt, gefangen genommen werden. Der Elefant ist also an seine Position gebunden. Eine einfache Berechnung des *Strongest Free Piece* würde das silberne Pferd auf c4 als die höchste, nicht gebundene Figur erkennen. Für Gold erscheint die Situation aussichtslos, da sowohl der goldene Hund auf b3, als auch der goldene Hase auf c5 und das goldene Pferd auf g7 gleichzeitig bedroht werden. Eine Sicherung aller drei Figuren ist schwer möglich.

Das nicht erkannte Problem tritt allerdings auf, sobald Gold seinen Elefanten auf c5 zieht und den goldenen Hasen auf b5. Der entsprechende Halbzug wäre folgender: 67g Ee6s Ee5w Rc5w Ed5w. Und resultiert in folgender Spielfeldsituation:

Abbildung 29: *Aufgabe der Hostage-Situation und Goal-Threat*

Unter anderem durch eine falsche Berechnung des Strongest-Free-Piece hat sich die Situation umgedreht. Gold hat nun eine starke Bedrohung der Zielerreichung durch den goldenen Hasen auf b5 geschaffen. Der silberne Elefant ist nun nicht nur nicht in der Lage, das goldene Pferd gefangen zu nehmen, sondern muss alle seine verfügbaren Schritte nutzen, um ein sofortiges Verlieren zu verhindern.

7.2.12 Force Distribution

Force Distribution ist ein umstrittenes Thema der Evaluierung. Die Evaluierung der Force Distribution sorgt dafür, dass die Figuren auf dem Spielfeld so angeordnet sind, dass eine gleichmäßige Aufteilung starker und schwacher Figuren entsteht. Dieser Umstand alleine jedoch birgt erhebliche Risiken, wenn der Gegner eine stark asymmetrische Aufstellung wählt oder alle starken Figuren nutzt, um eine Seite zu beherrschen. Bei einer gewählten gleichmäßigen Aufstellung sind zeitraubende Züge vonnöten, um eine gute Verteidigung aufzustellen oder einen Gegenangriff zu starten.

Eine andere Möglichkeit einer Evaluierung einer Force-Distribution ist es, diese an die Positionierung des Gegners anzupassen, um auf alle möglichen Angriffe reagieren zu können. Dadurch entsteht jedoch ein defensiv ausgerichtetes Spiel, welches immer einen Halbzug hinter einem offensiven Gegner her zieht und eher auf das Retten von Figuren ausgelegt ist.

Trotz der genannten möglichen Nachteile einer Force Distribution kann diese in kleinem Maße einige Vorteile schaffen. Zumindest, wenn die Evaluierung einer Force Distribution nicht höher wiegt, als die der anderen Komponenten der Evaluierung und eher kleine Zusatzpunkte vereinzelter Züge bewirkt, welche die Gesamtsituation des Spielfelds stärken.

7.2.13 Lemming Prevention

Die sogenannten Lemminge sind meist das Resultat einer Situation wie Hostage, Fork oder der Kontrolle eines gegnerischen Traps. Dabei besteht meist die Gefahr einer Gefangennahme einer mittleren oder hohen Figur durch den Gegner. Die einzige Möglichkeit eine sofortige Gefangennahme zu verhindern, besteht daraus, kleinere Figuren zur Verteidigung heranzuziehen, welche sich dabei selber in Gefahr bringen.

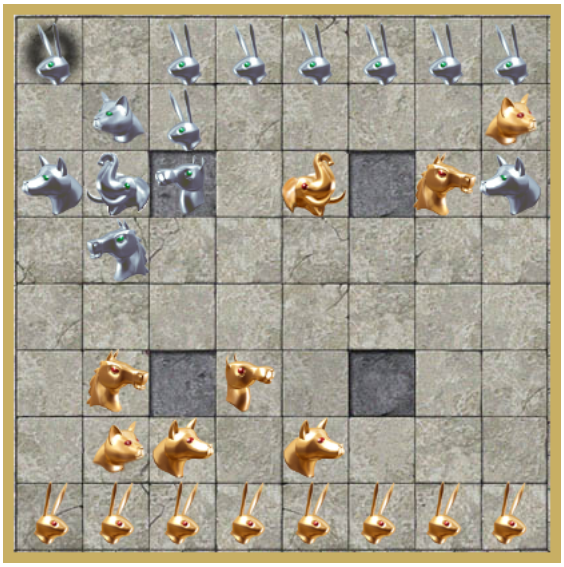


Abbildung 30: *Lemminge*

Im obigen Beispiel besteht die Gefahr, dass jederzeit der silberne Hund auf *h6* durch das goldene Pferd auf *g6* gefangengenommen wird. Die sofortige Möglichkeit einer Rettung besteht nur darin, die silbernen Hasen der achten Reihe als Verstärkung auf *f7* zu ziehen. Dadurch sind sie aber der sofortigen Gefangennahme ausgesetzt solange, bis keine Hasen mehr zur Verfügung stehen oder eine andere Lösung gefunden wurde.

Diese Lemming-Methode funktioniert vor allem bei Bots, da eine Evaluierung der Trapsituation ergibt, dass eine Gefangennahme des Hundes zum Beispiel -10 Punkte gibt, während die Gefangennahme eines Hasen nur -5 Punkte ergibt. Dadurch erscheint das Opfern des Hasen in dem Moment als strategisch sinnvoller, als den Hund zu opfern oder einen Gegenangriff zu starten.

Da Bots aber meist nicht in der Lage sind, weit genug in die Zukunft zu blicken, um dies zu verhindern, ist es möglich, auf diese Weise 2-5 Hasen und schlussendlich sogar den Hund gefangen zu nehmen, wenn keine andere Lösung für die Lemming-Situation gefunden wurde. Gerade durch den Umstand, dass die „Lemming"-Strategie eine Strategie ist, die sehr viele Züge im Voraus beinhaltet, ist es für Bots nur schwer frühzeitig zu erkennen und entsprechend zu evaluieren.

Je nach Art und Erfolg einer Implementierung der Fork, Hostage und Trapkontrolle können Lemminge schon hier vorgebeugt und frühzeitig verhindert werden. Trotz allem sollte bei der Evaluierung ein Bereich der Erkennung von Lemmingen gewidmet werden um der strategisch ungünstigen Situation vorzubeugen oder spätestens dann korrekt zu reagieren.

7.3 Dynamisch

Im Gegensatz zu der statischen Herangehensweise und ausschließlichen Evaluierung von Endknoten einer bestimmten Rekursionstiefe ist es auch möglich, eine teildynamische Evaluierung zu entwickeln. Die Idee hinter der Dynamik ist es, nicht nur Endzustände zu bewerten, sondern frühzeitig zu erkennen, wann bestimmte Pfade keinen Gewinn bringen und diese nicht weiter zu traversieren.

Die Funktionaltität der Evaluierung beruht auf dem Prinzip, dass jeder einzelne Knoten eines Suchbaumes unter gewissen Kriterien ausgewertet wird. Jedem rekursiven Aufruf innerhalb der Traversierung des Baumes wird eine Instanz einer speziellen Evaluierungsfunktion übergeben. Mittels dieser übergebenen Instanz wird an jeder Position im Baum der aktuelle Zustand und die Veränderung dieses zur vorherigen Position berechnet.

Die Situationserkennung beinhaltet sinnvollerweise eine Mustererkennung gewisser strategischer Elemente. Einige Beispiele strategischer Mustererkennung wäre Lemming Prevention, Gefangennahmen, Hostage, Frame und Fork-Situationen. Diese würden nicht erst in einem Endknoten des Suchbaumes erkannt und bewertet werden, sondern schon auf dem Weg dorthin.

Diese Möglichkeit der Evaluierung bietet zwei große Vorteile gegenüber der statischen Herangehensweise. Einmal kann die Evaluierung der tatsächlichen Endknoten sehr leichtgewichtig, einfach und effizient gehalten werden, da viele Muster und Aspekte schon erkannt und bewertet wurden. Der wichtigste Punkt jedoch ist, dass Cut-Off Möglichkeiten erzeugt werden können, wenn ein Halbzug auf lange Sicht nicht sinnvoll ist.

Ein Beispiel hierfür wäre, wenn der Elefant freiwillig in ein Trap geht (und dadurch kein sofortiger Gewinn entsteht). Ein normaler Alpha-Beta-Algorithmus wertet diesen Teilbaum vollständig aus, während der dynamische Ansatz den gesamten Teilbaum aus der Liste der möglichen Züge streichen kann, da die Opferung eines Elefanten nicht sinnvoll ist, wenn man damit nicht unmittelbar gewinnt.

Ein großer Nachteil des dynamischen Ansatzes ist jedoch, dass ein nicht zu unterschätzender Overhead entsteht, da nun nicht nur die Endknoten, sondern auch alle Knoten auf dem Weg dorthin evaluiert werden müssen. Dies kann unter Umständen zu einer Reduktion der Rekursionstiefe führen.

Eine weitere Entwicklung und Wettbewerbstauglichkeit dieser Methode bleibt zu analysieren. Unter Umständen lässt sich die Dynamik sehr leichtgewichtig gestalten und mit dem statischen Ansatz kombinieren, um mit wenig Overhead Cut-Offs zu erzeugen und das Programm stärker zu gestalten.

8 Tests

Durch die Vielfalt und die Menge an unterschiedlichen Möglichkeiten, die bei der Entwicklung eines Arimaa-Bots zum Einsatz kommen können, ist ein separates Testen der einzelnen Komponenten essentiell, bevor weitere Komponenten und Algorithmen auf diesen aufbauen.

Schon das Nachvollziehen der Ergebnisse einer normalen Alpha-Beta-Suche erfordert ein hohes Maß abstrakten Denkens. Wenn bei einer zusätzlichen Implementierung weiterer Komponenten wie Move-Ordering oder Transposition Tables nun Fehler auftreten oder Ergebnisse nicht den Erwartungen entsprechen, ist es oft hilfreich, Fehler aller alten Komponenten ausschließen zu können. Im Zuge der Ausarbeitung dieser Arbeit wurde ein Arimaa-Bot entwickelt, der einige der vorgestellten Algorithmen und Möglichkeiten implementiert hat.

Um eine Übersicht bezüglich der Effizienz und Performance der einzelnen Komponenten zu bekommen, sind alle Tests auf wenige spezielle Spielfeldsituationen und Rekursionstiefen festgelegt. Durch die identischen Rahmenbedingungen kann nun ein direkter Vergleich zwischen verschiedenen Komponenten gezogen werden. Der Arimaa-Bot wurde ausschließlich in Java entwickelt und kann über eine JVM (Java Virtual Machine) systemunabhängig ausgeführt werden.

8.1 Hardware und Systemumgebung

Das System, welches ausschließlich und für alle Testfälle genutzt wurde, besitzt die in der folgenden Tabelle dargestellten Systemattribute:

RAM Typ	DDR3 - 1333
RAM Kapazität	8GB
CPU Typ	AMD
CPU Kerne	4
CPU GHz	3.2
System	Unix Linux Mint 15 Cinnamon
Java Version	1.7.0_25

Tabelle 13: *Hardwareattribute und Systemumgebung*

8.2 Implementierte Algorithmen und Komponenten

Die folgenden Algorithmen, Komponenten und Erweiterungen wurden in den Arimaa-Bot implementiert:

- Grundalgorithmen
  - Minimax
  - NegaMax
  - Alpha Beta Pruning
  - Iterative Deepening mit Transposition Tables und Move-Ordering
- Algorithmische Erweiterungen
  - Dreifache Wiederholung eines Spielzustandes
  - Move-Ordering
  - Transposition Tables
  - Zobrist-Hash
  - Quiescence-Search
- Jegliche Form der statischen Evaluierung

Der Einfachkeit halber wird der NegaMax-Algorithmus bei den Tests nicht aufgeführt, da seine Auswertung identisch ist zu einem Minimax oder Alpha-Beta-Algorithmus.

8.3 Testfälle und Ergebnisse

Jeder hier aufgeführte Testfall wurde mehrere Male durchgeführt. Die Zeiten können wenige Sekunden von einander abweichen, so dass ein arithmetisches Mittel gebildet wurde.

Die abgebildete Spielsituation ist dem Spiel mit der Gameroom-ID *286183* entnommen und enstand nach dem 12. Zug mit Silber an der Reihe.

Arimaa-Notation: *ra7 rb8 rc8 rd8 re8 rf8 rg8 rh8 cc7 cf7 hg7 Ra6 db6 hd6 ef6 Mg6 mb5 Ec5 Cc6 Re5 Cf5 dg5 Dc4 Df4 Hb3 Rb2 Rd2 Hf2 Rh2 Rc1 Rf1 Rg1*

Spielfeld:

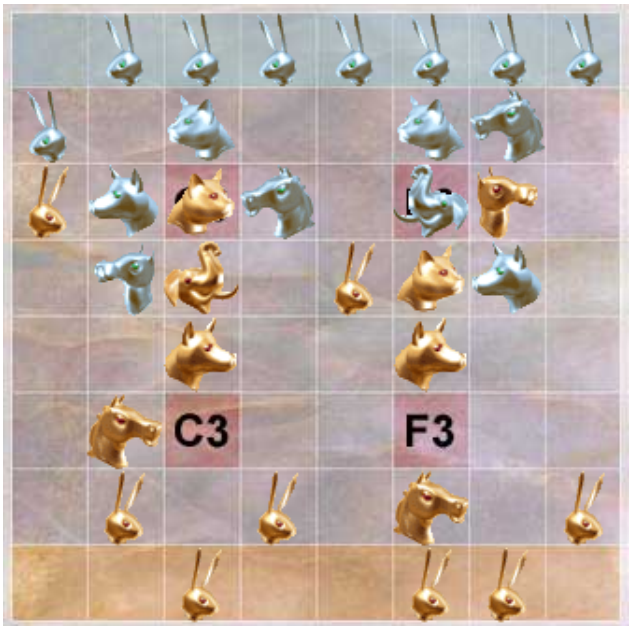


Abbildung 31: *Testaufstellung mit der Möglichkeit einer Kamel-Gefangennahme*

Der erste Test wurde mit einer Rekursionstiefe von $d = 6$ absolviert.

Algorithmus	**Zeit (Sekunden)**	**Anzahl Knoten**
Minimax	Unterbrochen nach 14332	194.751.498
$\alpha\beta$[2]	13	146.225
$\alpha\beta$ mit TT[3]	4	34.698
$\alpha\beta$ mit MO und TT	4	34.653
ID[4] mit $\alpha\beta$ und TT	6	66.078
ID mit $\alpha\beta$, TT und MO	6	60.653

Tabelle 14: *Testergebnisse des ersten Tests*

Das Resultat der Suche ist in Arimaa-Notation: *ef6w Mg6w dg5n Cf5e Mf6x.*

[2]Alpha Beta
[3]Transposition Table
[4]Iterative Deepening

Die Spielfeldsituation bietet einen entscheidenden Vorteil gegenüber vielen anderen Spielsituationen, der eine Suche sehr effizient und schnell gestalten kann. Durch die Möglichkeit der Gefangennahme des goldenen Kamels wird ein Halbzug mit extrem hohem Wert generiert, der dadurch sehr viele Cut-Offs erzeugen kann. Hier kann also ein normaler Alpha Beta-Algorithmus mit Memory schneller sein, als ein Iterative Deepening-Framework mit der Nutzung eines Alpha Beta-Algorithmus darin.

Der nächste Test bezieht sich auf identische Umstände, wie der vorherige Test mit einer erhöhten Rekursionstiefe von $d = 8$.

Algorithmus	**Zeit (Sekunden)**	**Anzahl Knoten**
$\alpha\beta$[5]	773	11.348.812
$\alpha\beta$ mit TT[6]	703	10.441.152
$\alpha\beta$ mit MO und TT	706	10.511.270
ID[7] mit $\alpha\beta$ und TT	754	10.864.160
ID mit $\alpha\beta$, TT und MO	575	8.233.738

Tabelle 15: *Testergebnisse des ersten Tests*

Das Resultat der Suche ist in Arimaa-Notation: *ef6w Mg6w dg5n Cf5e Mf6x.*

Das Ergebnis des gleichen Tests mit erhöhter Rekursionstiefe zeigt deutlich höhere und realistischere Zeiten auf. Zu beachten ist, dass jede Erhöhung der Rekursionstiefe die Anzahl zu evaluierender Knoten exponentiell wachsen lässt.

Auffällig ist zusätzlich, dass $\alpha\beta$ mit Transposition Tables und Move-Ordering offensichtlich keinen bedeutenden Vorteil bezüglich des gleichen Tests ohne Move-Ordering aufweist. Hierbei ist zu erwähnen, dass sich die implementierte Version des Move-Orderings ausschließlich auf Werte des Transposition Tables bezieht, als Optimierung für die Nutzung im Iterative Deepening Framework. Bei einer reinen $\alpha\beta$-Suche sind keine oder extrem wenig Knoten vorberechnet, so dass im Prinzip kein Move-Ordering stattfinden kann, jedoch entsteht ein gewisser Overhead durch den wiederholten Lookup.

In dem folgenden Test wird die Position nach dem 19. Zug von Gold im Spiel mit der Gameroom-ID *289569* als Testgrundlage genommen, welcher keine extremen Punkte durch sehr gute Züge zulässt. Der zu ziehende Spieler ist Silber.

[5] Alpha Beta
[6] Transposition Table
[7] Iterative Deepening

Arimaa-Notation: *rf8 ra7 rc7 dd7 re7 cf7 rg7 rh7 ra6 hb6 dd6 Ee6 Mf6 mg6 rh6 Ra5 Hb5 hd5 ef5 Rg5 Rh5 Ra4 Hd4 Df4 Cg4 Cb3 Dd3 Ra2 Rc2 Rf2 Rg1*

Spielfeld:

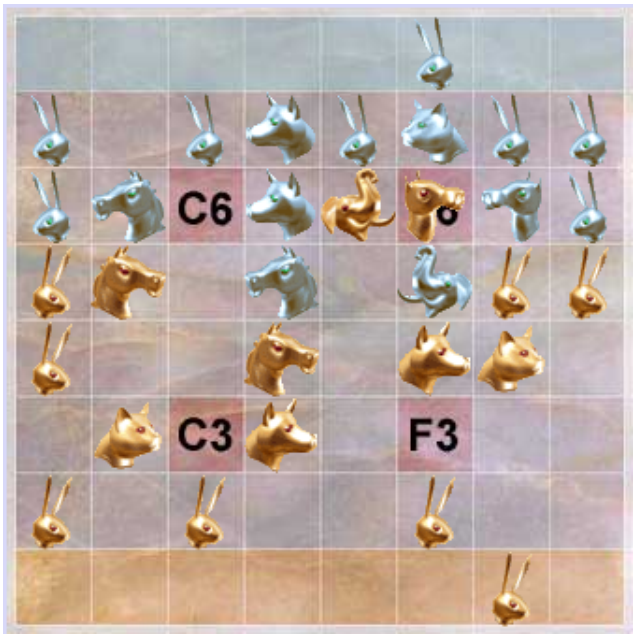


Abbildung 32: *Komplexere Testaufstellung*

In diesem Test werden alle Algorithmen mit einer Rekursionstiefe von $d = 8$ getestet, um die Ergebnisse zu verdeutlichen.

Algorithmus	**Zeit (Sekunden)**	**Anzahl Knoten**
$\alpha\beta$	1193	18.160.578
$\alpha\beta$ mit TT	690	10.472.271
$\alpha\beta$ mit MO und TT	579	8.595.256
ID mit $\alpha\beta$ und TT	729	10.981.461
ID mit $\alpha\beta$, TT und MO	469	7.543.041

Tabelle 16: *Testergebnisse einer komplexeren Spielfeldsituation*

Das Resultat der Suche ist in Arimaa-Notation: *rf8e cf7n re7e dd7e.*

Die Ergebnisse einer komplexeren Spielfeldsituation in der obigen Tabelle zeigen Ergebnisse, wie sie in vielen Spielsituationen in einem Spiel gegen hohe Spieler zu erwarten sind. Es kommt sehr selten zu evaluierten Werten, die stark von den weiteren Werten abweichen. Dadurch sind seltener und eher kleinere Cut-Offs zu erwarten, und die Suche nimmt längere Zeiten in Anspruch.

Die Testergebnisse spiegeln die Ergebnisse sehr vieler nicht aufgeführter Tests wieder, bei denen im Gesamtergebnis eine Nutzung eines Iterative Deepening Frameworks mit Alpha Beta, Transposition Tables und Move-Ordering konstant die besten Resultate erzielte.

8.4 CPU Sampling und Profiling

Dieser Abschnitt stellt ein CPU-Profiling der einzelnen fortgeschritteneren Algorithmen dar, welche in dem obigen Abschnitt im Test verwendet wurden. Das Profiling wurde ermittelt aus den Tests der folgenden Spielfeldsituation.

Arimaa-Notation: *rf8 ra7 rc7 dd7 re7 cf7 rg7 rh7 ra6 hb6 dd6 Ee6 Mf6 mg6 rh6 Ra5 Hb5 hd5 ef5 Rg5 Rh5 Ra4 Hd4 Df4 Cg4 Cb3 Dd3 Ra2 Rc2 Rf2 Rg1*

Spielfeld:

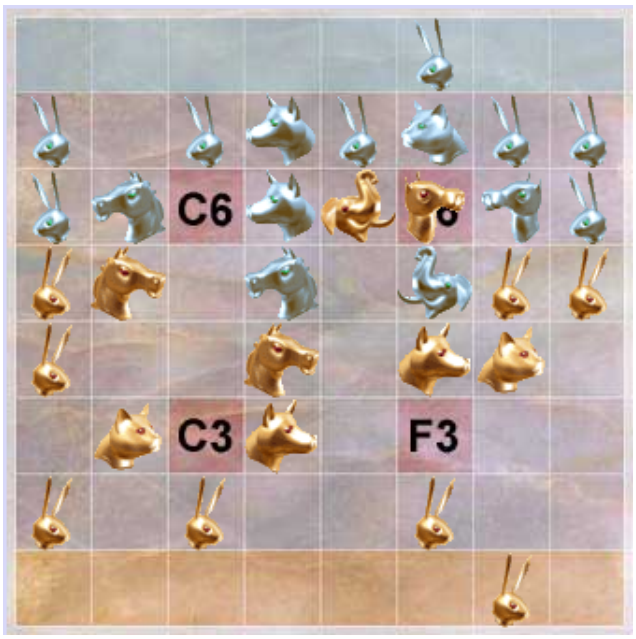


Abbildung 33: *Komplexere Testaufstellung*

Alpha Beta:

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
evaluation.SimpleEvaluation.**trapControlEvaluation** ()		44942 ms (55,4%)	44942 ms
evaluation.SimpleEvaluation.**evaluateBoardState** ()		8756 ms (10,8%)	8756 ms
board.Bitboard.**findPieceAt** ()		7433 ms (9,2%)	7433 ms
evaluation.SimpleEvaluation.**mobilityEvaluation** ()		4333 ms (5,3%)	4333 ms
board.Bitboard.**<init>** ()		2590 ms (3,2%)	2590 ms
evaluation.SimpleEvaluation.**rabbitWallEvaluation** ()		1836 ms (2,3%)	1836 ms
board.Bitboard.**getPieceSpecialMoveList** ()		1403 ms (1,7%)	1403 ms
board.MoveGenerator.**generateAllColorMoves** ()		1398 ms (1,7%)	1398 ms
evaluation.SimpleEvaluation.**getMoveCount** ()		1291 ms (1,6%)	1291 ms
memory.RepetitionCounter.**generateZobristHashForTable** ()		1095 ms (1,4%)	1095 ms
board.Bitboard.**pieceIsFrozen** ()		982 ms (1,2%)	982 ms
evaluation.SimpleEvaluation.**rabbitFreewayEvaluation** ()		900 ms (1,1%)	900 ms
evaluation.SimpleEvaluation.**lonelyCamelOrElephant** ()		895 ms (1,1%)	895 ms
evaluation.SimpleEvaluation.**frozenEvaluation** ()		605 ms (0,7%)	605 ms
evaluation.SimpleEvaluation.**passiveElephant** ()		511 ms (0,6%)	511 ms
evaluation.SimpleEvaluation.**camelPositioning** ()		490 ms (0,6%)	490 ms
evaluation.SimpleEvaluation.**elephantPositioning** ()		399 ms (0,5%)	399 ms
board.Bitboard.**getPieceMovementBitmap** ()		308 ms (0,4%)	308 ms
board.Bitboard.**getPullListAtPosition** ()		283 ms (0,3%)	283 ms
board.Bitboard.**getPushListAtPosition** ()		206 ms (0,3%)	206 ms
memory.RepetitionCounter.**getBoardHash** ()		109 ms (0,1%)	109 ms
board.Bitboard.**applyMove** ()		100 ms (0,1%)	100 ms
engine.AlphaBeta.**alphaBeta** ()		100 ms (0,1%)	100 ms
evaluation.SimpleEvaluation.**isFalseProtection** ()		99.6 ms (0,1%)	99.6 ms
board.Bitboard.**getColMap** ()		99.4 ms (0,1%)	99.4 ms
board.Bitboard.**cloneBitboard** ()		0.000 ms (0%)	0.000 ms
board.Bitboard.**applyMoveAndRemovePieces** ()		0.000 ms (0%)	0.000 ms
evaluation.SimpleEvaluation.**hostageSituation** ()		0.000 ms (0%)	0.000 ms
engine.AlphaBeta.**run** ()		0.000 ms (0%)	0.000 ms
engine.EngineTimeController.**run** ()		0.000 ms (0%)	0.000 ms
board.Bitboard.**removePiecesFromTraps** ()		0.000 ms (0%)	0.000 ms

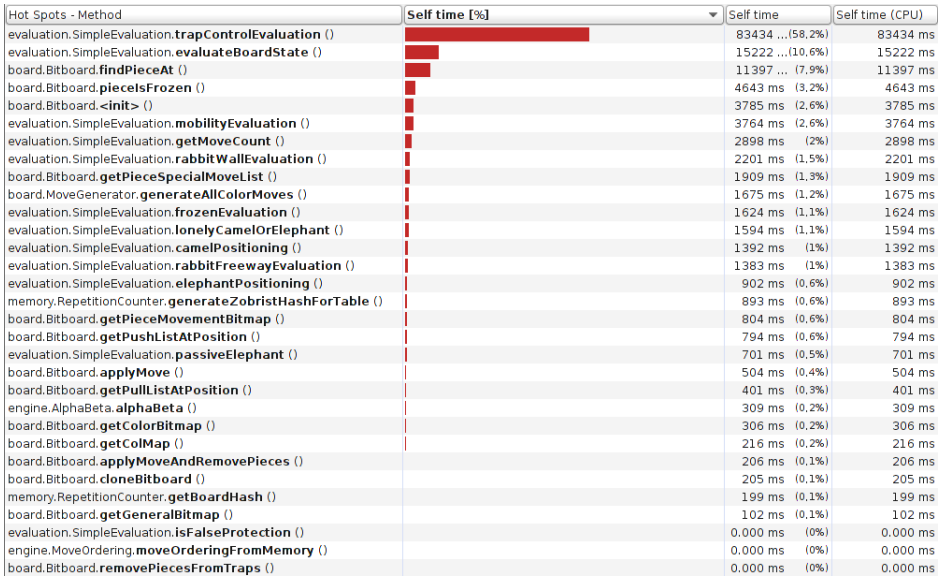
Abbildung 34: *CPU-Profiling bei Nutzung von $\alpha\beta$*

Alpha Beta mit Transposition Tables:

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
evaluation.SimpleEvaluation.**trapControlEvaluation** ()		1119... (59%)	111938 ms
board.Bitboard.**findPieceAt** ()		1848... (9,7%)	18480 ms
evaluation.SimpleEvaluation.**evaluateBoardState** ()		1722... (9,1%)	17222 ms
evaluation.SimpleEvaluation.**mobilityEvaluation** ()		8068 ... (4,2%)	8068 ms
evaluation.SimpleEvaluation.**rabbitWallEvaluation** ()		5321 ... (2,8%)	5321 ms
board.Bitboard.**pieceIsFrozen** ()		4293 ... (2,3%)	4293 ms
evaluation.SimpleEvaluation.**getMoveCount** ()		3803 ... (2%)	3803 ms
board.MoveGenerator.**generateAllColorMoves** ()		2790 ... (1,5%)	2790 ms
board.Bitboard.**<init>** ()		2690 ... (1,4%)	2690 ms
evaluation.SimpleEvaluation.**frozenEvaluation** ()		2480 ... (1,3%)	2480 ms
board.Bitboard.**getPieceSpecialMoveList** ()		1889 ... (1%)	1889 ms
evaluation.SimpleEvaluation.**lonelyCamelOrElephant** ()		1599 ... (0,8%)	1599 ms
memory.RepetitionCounter.**generateZobristHashForTable** ()		1400 ... (0,7%)	1400 ms
evaluation.SimpleEvaluation.**elephantPositioning** ()		1393 ... (0,7%)	1393 ms
evaluation.SimpleEvaluation.**camelPositioning** ()		1224 ... (0,6%)	1224 ms
evaluation.SimpleEvaluation.**rabbitFreewayEvaluation** ()		892 ms (0,5%)	892 ms
board.Bitboard.**getPushListAtPosition** ()		694 ms (0,4%)	694 ms
evaluation.SimpleEvaluation.**isFalseProtection** ()		613 ms (0,3%)	613 ms
board.Bitboard.**getPieceMovementBitmap** ()		599 ms (0,3%)	599 ms
evaluation.SimpleEvaluation.**passiveElephant** ()		592 ms (0,3%)	592 ms
board.Bitboard.**getPullListAtPosition** ()		405 ms (0,2%)	405 ms
board.Bitboard.**getHigherTypeMap** ()		309 ms (0,2%)	309 ms
board.Bitboard.**applyMove** ()		280 ms (0,1%)	280 ms
board.Bitboard.**getColorBitmap** ()		203 ms (0,1%)	203 ms
engine.AlphaBeta.**alphaBeta** ()		197 ms (0,1%)	197 ms
board.Bitboard.**applyMoveAndRemovePieces** ()		182 ms (0,1%)	182 ms
evaluation.SimpleEvaluation.**hostageSituation** ()		105 ms (0,1%)	105 ms
memory.Transposition.**addTableEntry** ()		100 ms (0,1%)	100 ms
board.Bitboard.**cloneBitboard** ()		98.8 ... (0,1%)	98.8 ms
memory.RepetitionCounter.**getBoardHash** ()		0.000... (0%)	0.000 ms
board.Bitboard.**removePiecesFromTraps** ()		0.000... (0%)	0.000 ms

Abbildung 35: *CPU-Profiling bei Nutzung von $\alpha\beta$ mit TT*

Alpha Beta mit Transposition Tables und Move-Ordering:

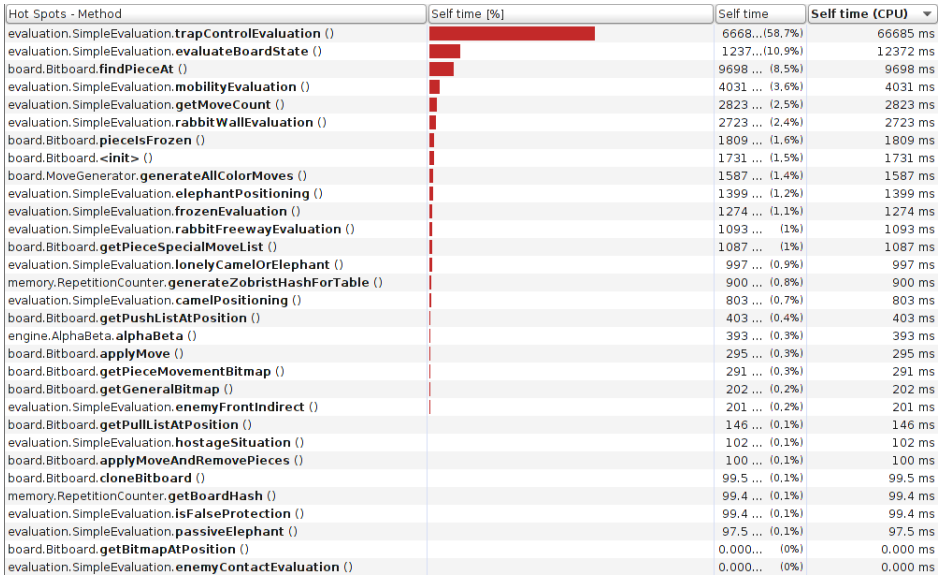


Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
evaluation.SimpleEvaluation.**trapControlEvaluation** ()		83434 ... (58,2%)	83434 ms
evaluation.SimpleEvaluation.**evaluateBoardState** ()		15222 ... (10,6%)	15222 ms
board.Bitboard.**findPieceAt** ()		11397 ... (7,9%)	11397 ms
board.Bitboard.**pieceIsFrozen** ()		4643 ms (3,2%)	4643 ms
board.Bitboard.**<init>** ()		3785 ms (2,6%)	3785 ms
evaluation.SimpleEvaluation.**mobilityEvaluation** ()		3764 ms (2,6%)	3764 ms
evaluation.SimpleEvaluation.**getMoveCount** ()		2898 ms (2%)	2898 ms
evaluation.SimpleEvaluation.**rabbitWallEvaluation** ()		2201 ms (1,5%)	2201 ms
board.Bitboard.**getPieceSpecialMoveList** ()		1909 ms (1,3%)	1909 ms
board.MoveGenerator.**generateAllColorMoves** ()		1675 ms (1,2%)	1675 ms
evaluation.SimpleEvaluation.**frozenEvaluation** ()		1624 ms (1,1%)	1624 ms
evaluation.SimpleEvaluation.**lonelyCamelOrElephant** ()		1594 ms (1,1%)	1594 ms
evaluation.SimpleEvaluation.**camelPositioning** ()		1392 ms (1%)	1392 ms
evaluation.SimpleEvaluation.**rabbitFreewayEvaluation** ()		1383 ms (1%)	1383 ms
evaluation.SimpleEvaluation.**elephantPositioning** ()		902 ms (0,6%)	902 ms
memory.RepetitionCounter.**generateZobristHashForTable** ()		893 ms (0,6%)	893 ms
board.Bitboard.**getPieceMovementBitmap** ()		804 ms (0,6%)	804 ms
board.Bitboard.**getPushListAtPosition** ()		794 ms (0,6%)	794 ms
evaluation.SimpleEvaluation.**passiveElephant** ()		701 ms (0,5%)	701 ms
board.Bitboard.**applyMove** ()		504 ms (0,4%)	504 ms
board.Bitboard.**getPullListAtPosition** ()		401 ms (0,3%)	401 ms
engine.AlphaBeta.**alphaBeta** ()		309 ms (0,2%)	309 ms
board.Bitboard.**getColorBitmap** ()		306 ms (0,2%)	306 ms
board.Bitboard.**getColMap** ()		216 ms (0,2%)	216 ms
board.Bitboard.**applyMoveAndRemovePieces** ()		206 ms (0,1%)	206 ms
board.Bitboard.**cloneBitboard** ()		205 ms (0,1%)	205 ms
memory.RepetitionCounter.**getBoardHash** ()		199 ms (0,1%)	199 ms
board.Bitboard.**getGeneralBitmap** ()		102 ms (0,1%)	102 ms
evaluation.SimpleEvaluation.**isFalseProtection** ()		0.000 ms (0%)	0.000 ms
engine.MoveOrdering.**moveOrderingFromMemory** ()		0.000 ms (0%)	0.000 ms
board.Bitboard.**removePiecesFromTraps** ()		0.000 ms (0%)	0.000 ms

Abbildung 36: *CPU-Profiling bei Nutzung von $\alpha\beta$ mit TT und MO*

Wie zu erwarten, hat das Move-Ordering keine (0 %) Auswirkung auf die Gesamtperformance, da es praktisch nicht verwendet werden kann, da keine vorberechneten Werte in dem Transposition Table vorhanden sind.

Iterative Deepening mit Transposition Tables:



Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
evaluation.SimpleEvaluation.**trapControlEvaluation** ()		6668... (58,7%)	66685 ms
evaluation.SimpleEvaluation.**evaluateBoardState** ()		1237... (10,9%)	12372 ms
board.Bitboard.**findPieceAt** ()		9698 ... (8,5%)	9698 ms
evaluation.SimpleEvaluation.**mobilityEvaluation** ()		4031 ... (3,6%)	4031 ms
evaluation.SimpleEvaluation.**getMoveCount** ()		2823 ... (2,5%)	2823 ms
evaluation.SimpleEvaluation.**rabbitWallEvaluation** ()		2723 ... (2,4%)	2723 ms
board.Bitboard.**pieceIsFrozen** ()		1809 ... (1,6%)	1809 ms
board.Bitboard.**<init>** ()		1731 ... (1,5%)	1731 ms
board.MoveGenerator.**generateAllColorMoves** ()		1587 ... (1,4%)	1587 ms
evaluation.SimpleEvaluation.**elephantPositioning** ()		1399 ... (1,2%)	1399 ms
evaluation.SimpleEvaluation.**frozenEvaluation** ()		1274 ... (1,1%)	1274 ms
evaluation.SimpleEvaluation.**rabbitFreewayEvaluation** ()		1093 ... (1%)	1093 ms
board.Bitboard.**getPieceSpecialMoveList** ()		1087 ... (1%)	1087 ms
evaluation.SimpleEvaluation.**lonelyCamelOrElephant** ()		997 ... (0,9%)	997 ms
memory.RepetitionCounter.**generateZobristHashForTable** ()		900 ... (0,8%)	900 ms
evaluation.SimpleEvaluation.**camelPositioning** ()		803 ... (0,7%)	803 ms
board.Bitboard.**getPushListAtPosition** ()		403 ... (0,4%)	403 ms
engine.AlphaBeta.**alphaBeta** ()		393 ... (0,3%)	393 ms
board.Bitboard.**applyMove** ()		295 ... (0,3%)	295 ms
board.Bitboard.**getPieceMovementBitmap** ()		291 ... (0,3%)	291 ms
board.Bitboard.**getGeneralBitmap** ()		202 ... (0,2%)	202 ms
evaluation.SimpleEvaluation.**enemyFrontIndirect** ()		201 ... (0,2%)	201 ms
board.Bitboard.**getPullListAtPosition** ()		146 ... (0,1%)	146 ms
evaluation.SimpleEvaluation.**hostageSituation** ()		102 ... (0,1%)	102 ms
board.Bitboard.**applyMoveAndRemovePieces** ()		100 ... (0,1%)	100 ms
board.Bitboard.**cloneBitboard** ()		99.5 ... (0,1%)	99.5 ms
memory.RepetitionCounter.**getBoardHash** ()		99.4 ... (0,1%)	99.4 ms
evaluation.SimpleEvaluation.**isFalseProtection** ()		99.4 ... (0,1%)	99.4 ms
evaluation.SimpleEvaluation.**passiveElephant** ()		97.5 ... (0,1%)	97.5 ms
board.Bitboard.**getBitmapAtPosition** ()		0.000... (0%)	0.000 ms
evaluation.SimpleEvaluation.**enemyContactEvaluation** ()		0.000... (0%)	0.000 ms

Abbildung 37: *CPU-Profiling bei Nutzung von ID mit $\alpha\beta$ und TT*

Iterative Deepening mit Transposition Tables und Move-Ordering:

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
evaluation.SimpleEvaluation.**trapControlEvaluation** ()		109863 ms (57,2%)	109863 ms
evaluation.SimpleEvaluation.**evaluateBoardState** ()		19572 ms (10,2%)	19572 ms
board.Bitboard.**findPieceAt** ()		16405 ms (8,5%)	16405 ms
evaluation.SimpleEvaluation.**mobilityEvaluation** ()		9301 ms (4,8%)	9301 ms
board.Bitboard.**getPieceSpecialMoveList** ()		5292 ms (2,8%)	5292 ms
board.Bitboard.**<init>** ()		5175 ms (2,7%)	5175 ms
evaluation.SimpleEvaluation.**getMoveCount** ()		4291 ms (2,2%)	4291 ms
board.Bitboard.**pieceIsFrozen** ()		3980 ms (2,1%)	3980 ms
evaluation.SimpleEvaluation.**rabbitWallEvaluation** ()		3388 ms (1,8%)	3388 ms
evaluation.SimpleEvaluation.**frozenEvaluation** ()		2672 ms (1,4%)	2672 ms
board.MoveGenerator.**generateAllColorMoves** ()		2215 ms (1,2%)	2215 ms
evaluation.SimpleEvaluation.**rabbitFreewayEvaluation** ()		1452 ms (0,8%)	1452 ms
board.Bitboard.**getPieceMovementBitmap** ()		1305 ms (0,7%)	1305 ms
evaluation.SimpleEvaluation.**camelPositioning** ()		1246 ms (0,6%)	1246 ms
memory.RepetitionCounter.**generateZobristHashForTable** ()		1189 ms (0,6%)	1189 ms
evaluation.SimpleEvaluation.**lonelyCamelOrElephant** ()		1115 ms (0,6%)	1115 ms
evaluation.SimpleEvaluation.**elephantPositioning** ()		907 ms (0,5%)	907 ms
engine.AlphaBeta.**alphaBeta** ()		707 ms (0,4%)	707 ms
board.Bitboard.**applyMove** ()		503 ms (0,3%)	503 ms
evaluation.SimpleEvaluation.**passiveElephant** ()		300 ms (0,2%)	300 ms
evaluation.SimpleEvaluation.**hostageSituation** ()		294 ms (0,2%)	294 ms
board.Bitboard.**cloneBitboard** ()		292 ms (0,2%)	292 ms
board.Bitboard.**applyMoveAndRemovePieces** ()		208 ms (0,1%)	208 ms
board.Bitboard.**getGeneralBitmap** ()		104 ms (0,1%)	104 ms
board.Bitboard.**getPullListAtPosition** ()		103 ms (0,1%)	103 ms
board.Bitboard.**getColorBitmap** ()		103 ms (0,1%)	103 ms
engine.MoveOrdering.**moveOrderingFromMemory** ()		100 ms (0,1%)	100 ms
memory.RepetitionCounter.**getBoardHash** ()		0.000 ms (0%)	0.000 ms
evaluation.SimpleEvaluation.**isFalseProtection** ()		0.000 ms (0%)	0.000 ms
board.Bitboard.**removePiecesFromTraps** ()		0.000 ms (0%)	0.000 ms
engine.EngineTimeController.**run** ()		0.000 ms (0%)	0.000 ms

Abbildung 38: *CPU-Profiling bei Nutzung von ID mit $\alpha\beta$, TT und MO*

Bei der Hinzunahme des Move-Orderings verändert sich bezüglich der Performance kaum etwas. Dem Profiling nach zu urteilen nutzt das Move-Ordering (engine.MoveOrdering.moveOrderingFromMemory()) gerade mal 0.1 % der genutzten CPU-Zeit.

Über alle verschiedenen Ergebnisse ist zu erkennen, dass unabhängig vom genutzten Algorithmus oder den verwendeten algorithmischen Erweiterungen ein sehr großer Teil der CPU-Zeit bei der Evaluierung des Spielfeldzustandes vergeht. Hier findet sich ein guter Ansatz für Performanceverbesserungen durch Optimierungen.

Eine genauere und detailreichere Analyse der prozentualen Verteilung der Methodenaufrufe ist schwer möglich, da das Profiling je nach Art der Suche und Spielfeldsituation in kleinem Rahmen oszilliert und sich nur grob (1-2 %) auf konkrete Werte einpendelt.

9 Vorstellung des Arimaa-Bots bot_Yeti_bsc

Das Resultat der vorliegenden Arbeit ist ein Programm welches in der Lage ist selbstständig Arimaa zu spielen. Diese Programme werden gemeinhin Bots genannt. Der Name des Bots ist „bot_Yeti_bsc“. Über das Onlineportal auf Arimaa.com können sowohl Menschen, als auch Bots gegeneinander antreten und spielen. Alle über die offizielle Seite gespielten Spiele werden gespeichert und sind jederzeit erneut abrufbar und einsehbar.

Abhängig der gespielten Spiele ergibt sich für jeden Spieler ein Rating. Bei jedem Spiel verändert sich das Rating beider Spieler abhängig vom Gewinn oder Verlust. Dabei spielen die Unterschiede des Ratings eine große Rolle bei der Anpassung dieser nach dem Spiel. Gewinnt der Spieler mit weniger Punkten, so bekommt er mehr Punkte für den Gewinn, als wenn der nach Punkten stärkere Spieler gewinnen würde. Das Rating-System ist vergleichbar mit dem ELO-Rating, welches beim Schach genutzt wird und variiert zwischen minimal ca. 1000 und maximal ca. 2600.

Alle von bot_Yeti_bsc gespielten Spiele und der jeweilige Ausgang der einzelnen Spiele sind auch unter folgendem Link einsehbar:

`http://arimaa.com/arimaa/gameroom/pastrecord.cgi?id=21768`

Das Profil mit Informationen über alle gespielten Spiele von bot_Yeti_bsc ist unter folgendem Link einsehbar:

`http://arimaa.com/arimaa/gameroom/playerpage.cgi?id=21768`

Bot_Yeti_bsc erreichte nach einer Reihe gespielter Spiele gegen andere Bots über das Onlineportal auf Arimaa.com ein Rating von ca. 1400. Dieses variiert jedoch bei jedem gespielten Spiel und erreichte Werte von unter 1200 und über 1500. In folgender Tabelle findet sich eine Zusammenfassung aller von bot_Yeti_bsc gespielten Spiele, sortiert nach der Anzahl der Spiele. Bei Spielen zwischen Bots mit einem Rating von unter ca. 1600 ist ein Gewinn oder Verlust meist abhängig von wenigen sehr guten Zügen oder Fehlern, welche der Bot nicht erkannt hat. So kann es sein, dass der schlechtere Bot auch Spiele gegen stärkere Bots gewinnen kann. Eine Gewinnchance an wenigen hundert ELO-Punkten festzumachen ist daher selten aussagekräftig.

Zusätzlich ist es sehr leicht möglich das Rating eines Spielers um mehrere hundert Punkte zu beeinflussen, indem tendentiell gegen schwächere Bots (Das eigene Rating ist höher) oder stärkere Bots gespielt wird (Das eigene Rating ist niedriger). Somit kann das Gameroome-Rating nur als möglicher Ansatzpunkt für die Feststellung der Stärke eines Spielers genommen werden.

Gegner	**R**[8]	**S**[9]	**G**[10]	**V**[11]	**G%**[12]
bot_Loc2007P1	1350	13	10	3	76.92
bot_Arimaazilla	1500	12	4	8	33.33
bot_Loc2006P2	1550	11	3	8	27.27
bot_Sharp2010P1	1600	5	0	5	0.00
bot_Marwin2010P1	1500	5	0	5	0.00
bot_GnoBot2010P1	1400	4	1	3	25.00
bot_Loc2005P2	1500	3	3	0	100.00
bot_OpFor2008P1	1450	3	1	2	33.33
bot_Loc2007P2	1500	3	0	3	0.00
bot_Briareus2011P1	1450	2	0	2	0.00
bot_Sharp2008P2	1500	2	0	2	0.00
bot_PragmaticTheory2010P1	1450	2	1	1	50.00
bot_ArimaaScoreP1	1000	2	2	0	100.00
bot_GnoBot2004CC	1400	1	1	0	100.00
bot_Clueless2008P1	1500	1	0	1	0.00
bot_Loc2006CC	1600	1	0	1	0.00
bot_Clueless2005P1	1600	1	0	1	0.00

Tabelle 17: *Spiele von bot_ Yeti_ bsc*

[8]Ungefähres durchschnittliches Rating
[9]Anzahl Spiele
[10]Gewonnen
[11]Verloren
[12]Gewonnen in %

10 Fazit und zukünftige Forschungsmöglichkeiten

Ziel der vorliegenden Studie war es, die Konzeption und Komplexität des strategischen Brettspiels Arimaa aufzuzeigen und mittels einer eigenen Implementierung eines Bots zu begründen.

Zu diesem Zweck wurden viele der grundlegenden Algorithmen, welche in den zur Zeit wettbewerbstauglichen Bots in Arimaa eingesetzt werden, untersucht und miteinander verglichen. Dabei wurde deutlich, wie komplex und aufwändig eine vollständige Berechnung eines Suchbaumes für Arimaa sein kann und wie nur geringe Rekursionstiefen und Steps selbst mit den fortschrittlichsten Algorithmen zu erreichen sind. Unter diesen Gesichtspunkten ist es ersichtlich, warum Menschen in diesem Spiel durch langfristige Strategien den Computern noch überlegen sind.

Jedoch verbleiben noch einige offene Punkte, in die eine weitere Forschung vielversprechende Ergebnisse liefern kann. So können eine gute Parallelisierung, Evaluierung und Reduktion des Branching Faktors die Rekursionstiefe weiter vertiefen. Auch mit Hilfe einer ausgeprägten selektiven Quiescence-Search lässt sich in speziellen Situationen die Suche weiter vorantreiben. Weitere Gebiete der künstlichen Intelligenz stehen offen und bedürfen einer gründlichen Auseinandersetzung, um eine mögliche Eignung für Arimaa-Bots zu erschließen.

Abschließend kann zusammengefasst werden, dass eine Entwicklung von Arimaa Bots weit von ihrem Zenith entfernt ist. Trotz der komplexen und vielschichtigen Grundvoraussetzungen, der wenigen Jahre an Entwicklungserfahrung und der verhältnismäßig kleinen Anzahl an aktiven Entwicklern haben Arimaa Bots ein doch überraschend hohes Niveau erreicht, welches nur wenige hundert ELO-Punkte unter den besten Menschen liegt.

So wird das Wettrennen zwischen Mensch und Computer auch die nächsten Jahre im Fachgebiet Arimaa weitergehen und mit Sicherheit viele spannende Entdeckungen und Überraschungen bereit halten.

Literatur

[1] Aberent. Transposition Table and Zobrist Hashing. http://www.chessbin.com/post/transposition-table-and-zobrist-hashing.aspx, February 2010.

[2] Mark Brockington. An Implementation of the Young Brothers Wait Concept. University of Alberta, 1994.

[3] Jeroen W.T. Carolus. *Alpha-Beta with Sibling Prediction Pruning in Chess*. University of Amsterdam, 2006.

[4] Gobet & Clarkson. Chunking Theory. http://snitkof.com/cg156/chunkingtheory.php, 2004.

[5] Christ-Jan Cox. Analysis and implementation of the game Arimaa. Universiteit Maastricht, Maastricht ICT Competence Centre, 2006.

[6] Van-Dat Cung. Exploration Parallele d'Arbres de Jeux Minimax. Universite de Versailles-St. Quentin, O.J.

[7] Eldar Erathis. Time Delays. http://code.google.com/p/simplechessclock/wiki/TimeDelays, 2011.

[8] Uriel Feige. Algorithms for Computing Solution Concepts in Game Theory. Weizmann Institute, 2008.

[9] David Fotland. Building a World Champion Arimaa Program. Smart Games, 2004.

[10] Frederic Friedel. Pictures from the Big Apple. http://en.chessbase.com/post/pictures-from-the-big-apple, 2003.

[11] Didier Fuchs. Global Algebric Material Evaluator For Arimaa Game, 2010.

[12] Didier Fuchs. Global Evaluator of Material for Arimaa Gem, 2010.

[13] Didier Fuchs. Holistic Evaluator of Remaining Duels, 2010.

[14] Didier Fuchs. Holistic Material Evaluator for Arimaa game, 2010.

[15] Didier Fuchs. Arimaa Positional Evaluator, 2011.

[16] Didier Fuchs. Recursive Evaluator of Material, 2011.

[17] Prometeus GmbH. Top 500 Supercomputer. http://www.top500.org/list/1997/06/300/?page=3, Prometeus GmbH, June 1997.

[18] Brian Greskamp. Parallelizing a Simple Chess Program, 2003. ECE412.

[19] Brian Haskin. A Look at the Arimaa Branching Factor. http://arimaa.janzert.com/bf_study/, 2006.

[20] Christof Heymann. Minimax und Alpha-Beta-Pruning. Technical report, hochschule coburg – university of applied sciences, O.J.

[21] Christof Heymann. MiniMax und Alpha-Beta-Pruning. Hochschule Coburg, O.J.

[22] Tomas Hrebejk. Arimaa challenge - static evaluation function. Department of Theoretical Computer Science and Mathematical Logic, 2013.

[23] Gerd Isenberg. ABDADA. `http://chessprogramming.wikispaces.com/ABDADA`, 2013.

[24] Gerd Isenberg. Shared Hash Table. `http://chessprogramming.wikispaces.com/Shared+Hash+Table`, 2013.

[25] Petr Rockai Jiri Barnat. Shared Hash Tables in Parallel Model Checking. Paculty of Informatics, Masaryk University, 2007.

[26] Martin J.Osborne and Ariel Rubinstein. *A Course in Game Theory*. The MIT Press, 1994.

[27] Fritz Juhnke. *Chess Reborn Beyond Computer Comprehension*. Flying Camel Publications, June 2009.

[28] Garry Kasparov. Garry Kasparov begs to differ... `http://en.chessbase.com/post/garry-kasparov-begs-to-differ-`, December 2002.

[29] John Kelly. Top 10 Mind-bending Strategy Games. `http://entertainment.howstuffworks.com/leisure/brain-games/10-mind-bending-strategy-games8.htm`, 2010.

[30] François Dominic Laramée. Chess Programming Part II: Data Structures. `http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-ii-data-structures-r1046`, June 2000.

[31] Valavan Manohararajah. Parallel Alpha-Beta Search on Shared Memory Multiprocessors. University of Toronto, 2001.

[32] Samuel John Odell Miller. Researching and Implementing a Computer Agent to Play Arimaa. University of Southampton, 2009.

[33] Feng-hsiung Hsu Murray Campbell, A. Joseph Hoane Jr. Deep Blue. Technical report, Elsevier - Artificial Intelligence, 2001.

[34] Eric Patton. Humans v. Computers In Chess - Symbiosis Of The Two Is Superior. http://colonyofcommodus.wordpress.com/2012/06/06/humans-v-computers-in-chess-symbiosis-of-the-two-are-superior/, 2012.

[35] Aske Plaat. Research re:search & re-search. Erasmus University Rotterdam, 1996.

[36] Aske Plaat. MTD(f) algorithm. Erasmus University Rotterdam, 1997.

[37] IBM press release. Deep Blue. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/, IBM, February 2011.

[38] Tord Romstad. The Art of Evaluation. http://www.talkchess.com/forum/viewtopic.php?topic_view=threads&p=135133&t=15504, 2007.

[39] Dr. Afşar Saranli, Stuart Russel, and Peter Norvig. *Artificial Intelligence: A Modern Approach, 2nd Ed.* Pearson Higher Education, Inc., Upper Saddle River, 2010.

[40] Andre Schulz. Kramnik gegen Deep Fritz: Das letzte Match Mensch gegen Maschine? http://www.spiegel.de/netzwelt/tech/kramnik-gegen-deep-fritz-das-letzte-match-mensch-gegen - maschine-a-450147.html, 2006.

[41] Daaim Shabazz. Kasparov & Deep Junior fight to 3-3 draw. http://www.thechessdrum.net/tournaments/Kasparov-DeepJr/, 2003.

[42] Tsan sheng Hsu. Transposition Table, History Heuristic, and other Search Enhancements. http://www.iis.sinica.edu.tw/ tshsu, 2013.

[43] Peter Norvig Stuart Russell. *Artificial Intelligence A Modern Approach, Third Edition.* Pearson Education, Inc., 2010.

[44] Omar Syed. The Arimaa Challenge. http://arimaa.com/arimaa/challenge/, 1999.

[45] Omar Syed. The Creation of Arimaa. http://arimaa.com/arimaa/about/, 1999.

[46] Omar Syed. The 2014 Arimaa Challenge. http://arimaa.com/arimaa/challenge/2014/, 2013.

[47] Omar Syed. The Arimaa World Championship. http://arimaa.com/arimaa/wc/, 2013.

[48] Gerhard Trippen. Plans, Patterns and Move Categories Guiding a Highly Selective Search. The University of British Columbia, 2009.

[49] Vipin Kumar V. Nageshwara Rao. Parallel Depth First Search, Part I: Implementation. University of Austin, Texas, O.J.

[50] Nick Wedd. Computer Go - Past Events. http://www.computer-go.info/events/index.html, 2013.

[51] Melinda Wenner. Brainy Gifts. www.ScientificAmerican.com/Mind, 2009.

[52] By Wikipedians. Chess variants. http://pediapress.com/books/show/chess-variants-by-wikipedians/, 2014.

[53] Aske Plaat Wim Pijls, Arie de Bruin. A theory of game trees, based on solution trees. Erasmus University Rotterdam, O.J.

[54] David Jian Wu. Move Ranking and Evaluation in the Game of Arimaa. Harvard College, Cambridge, Massachusetts, March 31, 2011.

[55] T.A. Marsland Yaoqing Gao. Multithreaded Pruned Tree Search in Distributed Systems. University of Alberta, O.J.

[56] Haizhi Zhong. Building a Strong Arimaa-playing Program. University of Alberta, 2005.