ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Eine Dokumentation für das Modul Projektarbeit
Projekt “Schachengine”

vorgelegt von

Johannes Lux und Moritz Schönenberger

Modul betreut von

Prof. Dr. Thomas Kretschmer

Saarbrücken, 20. Oktober. 2022

Zusammenfassung

Inhalt dieser Dokumentation ist die Durchführung des Moduls “Projektarbeit” in der Gruppe “Schach-Engine”. Im Laufe des Moduls haben wir eine eigene Schach-Engine entworfen und implementiert. Danach haben wir sie optimiert und getestet mit dem Ziel, dass sie innerhalb von 30 Sekunden einen möglichst guten Zug berechnet.

Dieses Dokument enthält sowohl Erläuterungen zur Architektur der Engine und zum Ablauf des Projekts als auch zu den verwendeten Testkonzepten. Es werden die wichtigsten Modellklassen sowie die einzelnen Komponenten erklärt und die Änderungen, die sich im Laufe des Projekts ergeben haben, werden nachgezeichnet. Die Erklärungen erfolgen teilweise an Codebeispielen.

Inhaltsverzeichnis

1 Aufgabenstellung

Ziel des Projektes ist es, ein Programm zu schreiben, das mit Hilfe einer externen Benutzeroberfläche Schach spielen kann (im Folgenden Engine oder Schach-Engine genannt). Die Engine muss über das Universal Chess Interface mit der Benutzeroberfläche kommunizieren. Um zu spielen, muss die Engine zuerst zu einer gegebenen Stellung alle erlaubten Züge generieren und dann aus diesen Zügen mithilfe eines Spielbaumes den besten Zug auswählen. Dieser Zug muss dann mit dem Universal Chess Interface zurückgegeben werden.

Die folgende Aufzählung erhält die weiteren in Abstimmung mit Professor Kretschmer festgelegten Kriterien, die von der Schach-Engine erfüllt werden müssen:
Die Engine…

- …spielt Schach gemäß der FIDE-Regeln
- …ist jederzeit über UCI erreichbar, auch während der Berechnungen
- …wählt bei Bauernumwandlungen die gemäß Stellung beste Figur aus
- …soll möglichst gut Schach spielen
- …ist in der Lage, unter Zeitbeschränkung von 30 Sekunden einen Zug zu berechnen

2 Verwendete Werkzeuge

Zur Erstellung der Engine:

- Java - Programmiersprache, in der die Engine geschrieben ist
- JUnit - Testen der Engine
- Maven - Paketverwaltung und Continuous Integration
- Git - Versionsverwaltung
- GitHub - Hosting und Continuous Integration
- Lichess Analysebrett - Visualisierung von FEN-Strings und Qualitätsprüfung von Zügen, die die Engine berechnet hat
- Stockfish - Generieren von legalen Zügen für Testfälle

Zur Erstellung der Dokumentation:

- LaTeX- zur Setzung des Texts

- diagrams.net - für alle Grafiken, die in der Dokumentation verwendet werden

3 Installation der Engine

3.1 Kompilieren

Voraussetzung um das Programm kompilieren zu können ist, dass Maven auf dem Computer installiert ist (verfügbart unter `https://maven.apache.org/install.html`). Das Kompilieren erfolgt mit dem Befehl `mvn package` ausgeführt im Root-Ordner des geklonten Git-Repositories.

3.2 Einbinden in Arena

- In der Menüleiste von Arena unter dem Punkt `Motoren` die Option `Neuen Motor installieren` auswählen
- Die kompilierte .jar-Datei im Unterordner `target` des geklonten Git-Repositories auswählen (eventuell muss der Dateityp im Auswahlfenster auf .jar umgestellt werden, damit die Datei angezeigt wird)
- Bei der Frage nach dem Typ der Engine `UCI` wählen
- Zu `Motoren` > `Verwalten` gehen und den Reiter `Details` auswählen
- Die Engine SchachMotor auswählen und bei der Drop-Down-Liste `Typ` den Wert `UCI` auswählen

4 Kommunikation via Universal Chess Interface

Das Universal Chess Interface (UCI) ist ein Protokoll, mit dem Schach-Engines mit Benutzeroberflächen wie beispielsweise Arena kommunizieren können. Eingehende Befehle werden von dem Standard-Input des Engine-Prozesses aus (im Folgenden mit `stdIn` bezeichnet) gelesen, ausgehende Befehle werden auf den Standard-Output des Engine-Prozesses (im Folgenden mit `stdOut` bezeichnet) geschrieben. Eine umfangreiche Dokumentation zu UCI befindet sich auf folgender Webseite: `https://www.shredderchess.com/chess-features/uci-universal-chess-interface.html`.

4.1 Eingehende Kommunikation

4.1.1 Tokenizer

Der `UCITokenizer` bekommt ein von `stdIn` ausgelesenes UCI-Kommando als String übergeben. Danach trennt der Tokenizer den String anhand von Leerzeichen und schreibt die einzelnen Substrings in ein Array. Über dieses Array iteriert er dann und baut währenddessen einen Syntaxbaum auf, dessen Wurzel das erste Token im Kommando ist, das keine Konstante ist. Ein Token ist dann eine Konstante, wenn es nicht zu der Menge der für UCI in der Richtung Oberfläche zu Engine definierten Befehle gehört.

Die Knoten im Syntaxbaum sind alle Instanzen der `Command`-Klasse. Diese Klasse enthält einen `CommandType` (alle für UCI in der Richtung Oberfläche zu Engine definierten Befehle und der Typ `CONSTANT`). Außerdem haben Objekte der Klasse `Command` einen Verweis auf ihren Elternknoten, eine Liste mit ihren Kindknoten und einen String, in dem sie zusätzliche Daten speichern können (beispielsweise die Werte von Konstanten).

Bemerkenswert ist hier noch die Funktion `glueFenTogether()`. Diese existiert, da ein übergebener FEN-String selbst auch Leerzeichen enthält und deswegen beim Aufteilen des Input-Strings in seine Einzelteile zerlegt wird. Da die folgenden Verarbeitungsschritte allerdings erwarten, dass der eingegebene FEN-String als zusammenhängender String übermittelt wird, ist es notwendig, ihn im Tokenizer wieder zusammenzusetzen, nachdem er aufgeteilt wurde.

4.1.2 Parser

Es gibt verschiedene Parser im Projekt, wobei `UCIParserAlphaBetaPruning` die Produktivimplementierung ist. Diese Klasse hat die öffentliche, statische Methode `executeCommand()`, die einen UCI-Befehl als Input bekommt, der bereits vom Tokenizer verarbeitet wurde. Das Herzstück von `executeCommand()` ist ein großes Switch-Caste-Statement, das alle Werte des Enums `CommandType` abbildet. Hier wird die Logik implementiert, die ausgeführt wird, wenn ein Kommando dieses Typs eingelesen wird. Im Falle eines `position startpos` wird beispielsweise das interne, aktuelle Spielbrett in die Startstellung gebracht, während ein `go`-Kommando dazu führt, dass ausgehend von der intern gespeicherten Stellung der beste Folgezug berechnet wird.

Außerdem beinhaltet der Parser die Hilfsfunktion `isTheSameGame()`, die bestimmen kann, ob zwei von der Benutzeroberfläche übermittelte Züge zur selben Partie gehören und aufeinander folgen. Dies ist eine Optimierung, da Arena die erhaltenen Züge immer ausgehen von der Startposition übermittelt (also `position startpos moves` m_1 ... m_{2n} für ein Spiel mit n Zügen und 2n Halbzügen). Mithilfe von `isTheSameGame()` kann überprüft werden, ob die interne Stellung vom Start aus aufgebaut werden muss oder ob nur die letzten beiden Halbzüge auf die interne Repräsentation angewendet werden müssen.

4.2 Ausgehende Kommunikation

Um die Implementierung etwas modularer zu halten gibt es eine Klasse namens `UCIOperator`. Diese Klasse besitzt statische Methoden, mit denen sie alle für UCI in der Richtung Engine zu Oberfläche definierten Befehle auf `stdOut` schreiben kann. Befehle, die Parameter enthalten (beispielsweise `bestmove [move]`), werden von Methoden mit entsprechenden Argumenten implementiert.

5 Repräsentation einer Stellung

Die Repräsentation von Stellungen wird übernommen von der Klasse `Position`. Sie enthält alle Informationen, die auch in einem FEN-String enthalten sind und noch ein paar zusätzliche Informationen. Diese werden aus dem FEN-String berechnet und dann gespeichert, damit sie nicht mehrmals neu berechnet werden müssen. Im Lauf des Projekts haben wir an dieser Klasse einige Veränderungen vorgenommen.

5.1 Ursprüngliches Konzept

In ihrer ersten Form hatte die Klasse alle Informationen, die auch in einem FEN-String enthalten sind und zusätzlich noch zwei Boolean-Felder, um abzuspeichern, ob der weiße und der schwarze König im Schach stehen. Die Repräsentation des Spielbretts war noch nicht von der Stellung abgekapselt und war stattdessen ein zweidimensionales 8×8-Array bestehend aus `Piece`-Objekten. Ein `Piece`-Objekt beschreibt eine Figur auf dem Spielbrett, es enthält den Figurentyp und die Farbe der Figur. Jedes der 64 Felder des Spielbretts wurde einem Feld des Arrays zugeordnet. War das Feld auf dem Spielbrett leer, so wurde das korrespondierende Feld im Array auf `null` gesetzt; ansonsten wurde das korrespondierende Feld im Array mit einem `Piece`-Objekt belegt, das die Figur auf dem Feld beschreibt.

5.2 Abkapselung des Spielbretts

Als wir die erste Version von Alpha-Beta-Pruning ausprobierten, stellte sich heraus, dass es ein großes Platzproblem gab. Mit zunehmender Tiefe des Spielbaums wurden große Mengen an `Position`-Objekten erzeugt, die den Speicherbedarf der Engine stark aufblähten. Teilweise wurde sogar so viel Speicher angefordert, dass die Java-VirtualMachine keinen Speicher mehr allokieren konnte und die Engine stoppen musste.

Als Reaktion darauf entschlossen wir uns, die Klasse `Position` soweit wie möglich zu verkleinern und damit bei der Repräsentation des Spielbretts anzufangen. Um den Wechsel der Darstellung des Spielfeldes einfacher zu gestalten, haben wir als ersten Schritt das Interface `Board` gebaut. Danach sind wir den Code durchgegangen und haben alle direkten Zugriffe auf das Array mit Zugriffen durch `getPieceAt()` und `setPieceAt()` ersetzt. Die bisherige Implementierung mit dem Array von `Piece`-Objekten haben wir in eine eigene Klasse namens `ArrayBoard` verschoben, die das `Board`-Interface implementiert. Statt einem `Piece`-Array enthält `Position` seitdem ein `Board`-Objekt, um das Spielbrett abzubilden.

Der mit Abstand größte Faktor dabei, das Speicherproblem zu lösen, war allerdings nicht die Speicheroptimierung der `Position`-Klasse. Es war eine Änderung der Bewertungsalgorithmen, die dafür sorgt, dass Knoten im Spielbaum aus dem Hauptspeicher gelöscht werden, nachdem sie ausgewertet wurden. Näheres zu dieser Änderung findet sich in Abschnitt 10.1.3.

5.3 Schacherkennung in `Position`

Um schnell abfragen zu können, ob einer der Könige sich gerade im Schach befindet, speicherten `Position`-Objekte in ihrer ursprünglichen Form in den Variablen `blackInCheck`

und `whiteInCheck` diese Informationen ab. Um diese Variablen zu füllen, werden im Konstruktor von `Position` zwei Attack Maps erstellt (siehe hierzu Kapitel 7). Die Werte an den Koordinaten der Könige haben wir herausgelesen und gespeichert für die Abfrage, ob der eigene König im Schach steht. Außerdem haben wir auch die Attack Maps selbst gespeichert, damit wir nachschauen konnten, ob einer der Könige im Falle einer Rochade über ein bedrohtes Feld ziehen müsste. Später haben wir allerdings anstatt die zwei Attack Maps komplett abzuspeichern nur noch 8 Boolean-Werte abgespeichert - für beide Seiten jeweils die beiden Felder in der langen und der kurzen Rochade, die der König passieren muss.

5.4 Bitcodierung der Boolean-Variablen

Als wir so weit waren, dass wir mit unserem Spielbaum in vielen Stellungen Tiefe 7 in 30 Sekunden erreichen konnten, haben wir gemerkt, dass das bedeutet, dass bei einer Berechnung mehrere Millionen Knoten ausgewertet werden. Daraus folgt, dass also auch mehrere Millionen `Position`-Objekte instanziiert werden müssen. Um die Klasse so weit wie möglich zu optimieren haben wir die 15 Boolean-Variablen in `Position` (4 für die Rochaderechte, 8 für die Felder, die während Rochaden von Königen passiert werden müssen, 2 für die Information, ob die beiden Könige im Schach stehen und 1, um zu markieren, welche Farbe als nächstes ziehen darf) in zwei Bytes codiert. Hierfür haben wir die Hilfsfunktion `setByteEncodedBoolean()` geschrieben, die ein Byte bekommt, den Index des zu setzenden Bits und den Wert, auf den das Bit gesetzt werden soll.

Zum Auslesen setzen wir Hilfsfunktion `getByteEncodedBoolean()` ein, die das Byte bekommt, in dem die gewünschte Information steht und den Index des Bits, das die gewünschte Information codiert. Diese Hilfsfunktion gibt `true` aus, wenn das Bit gesetzt ist und `false`, wenn nicht.

Da für die Variablen sowieso schon öffentliche Getter und Setter genutzt wurden waren nur wenige Schritte nötig, um den Wechsel von Booleans auf Bytes möglich zu machen. Zuerst musste sichergestellt werden, dass auch innerhalb von `Position` immer die Getter und Setter genutzt werden, um mit den Variablen zu interagieren. Als nächstes wurden zwei Byte-Klassenvariablen zu `Position` hinzugefügt, um die Boolean-Werte zu speichern. Danach wurde dann die Implementierung der Getter und Setter ausgetauscht, sodass diese Funktionen nicht mehr direkt mit den Boolean-Variablen in `Position` interagieren. Stattdessen rufen die Getter jetzt `getByteEncodedBoolean()` und die Setter `setByteEncodedBoolean()` auf. Der letzte Schritt war dann, die Boolean-Variablen in `Position` zu löschen.

Seitdem sind die 8 rochaderelevanten Felder in einem Byte abgespeichert und die 7 restlichen Wahrheitswerte befinden sich in dem anderen Byte.

5.5 Alternative Implementierungen von `Board`

Wir haben uns überlegt, ob wir BitBoards nutzen, aber wir wollten zunächst noch nicht mit Bitmasking arbeiten. Deshalb haben wir uns für einen Kompromiss entschieden, bei dem das Spielbrett immer noch durch ein 8×8-Array repräsentiert wird aber die einzelnen Felder des Arrays nicht mit `Piece`-Objekten sondern mit Bytes gefüllt sind. Die Implementierung dieses Konzeptes befindet sich in der Klasse `ByteBoard`. Jeder Figurentyp-Farbe-Kombination wurde dann ein Zahlenwert zugewiesen, wodurch die eindeutige Zuordnung möglich war.

Ein Schritt, der starke Performance-Verbesserungen zur Folge hatte, war, aus dem 8×8-Array ein eindimensionales Array der Größe 64 zu machen. Damit wurde mit jedem Zugriff auf das Board die Zugriffszeit halbiert, was insgesamt zu einer 20% schnelleren Berechnung geführt hat.

Der letzte Schritt, den wir bei der Spielfeldrepräsentation unternommen haben, um sie zu optimieren, ist, die Größe des Byte-Arrays zu halbieren. Dies ist möglich, da die höchste Zahl, die eine Figur codiert, die Zahl 13 ist. Der Grund dafür ist, dass die Zahlen von 1-6 für die verschiedenen weißen Figuren stehen, 7 eine exklusive Grenze zwischen den Farben darstellt und 8-13 für die verschiedenen schwarzen Figuren stehen.

Um die Zahl 13 darzustellen, benötigt man nur 4 der 8 Bits, die ein Byte ausmachen, also ist es möglich, zwei Figuren in einem Byte zu speichern. Somit konnten wir die Größe des Arrays auf 32 verringern, indem wir jeweils zwei Felder des Spielbretts in einem Byte codiert haben. Die Felder in den Linien a, c, e und g werden somit immer durch die vorderen 4 Bits ihres jeweiligen Bytes repräsentiert, die Felder in den Linien b, d, f und h immer durch die hinteren 4 Bits.

6 Aufbau und Funktionsweise des Zuggenerators

6.1 Legale oder pseudolegale Züge

Ein Zug heißt pseudolegal, wenn er zwar mit den Zugregeln der bewegten Figur konform ist (also beispielsweise ein Läufer wurde diagonal gezogen) aber aus einem anderen Grund nicht erlaubt ist (zum Beispiel weil beim Ausführen dieses Zuges der eigene König im Schach zurückgelassen werden würde).

Wir haben uns dazu entschieden, unseren Zuggenerator so zu konstruieren, dass er nur legale Züge zurückgibt und pseudolegale Züge verwirft.

Dies hat allerdings dazu geführt, dass unsere ursprüngliche Idee für das Erkennen eines Matts nicht funktioniert. Angedacht war hier nämlich, dass wir dem König einen sehr hohen Materialwert verleihen und das Schlagen des Königs erlauben. Somit sollte die Engine dann ein Matt dadurch erkennen, dass in einem zukünftigen Zug ein König geschlagen wird und der Materialwert dieser Stellung entsprechend hoch oder niedrig ist. Es hat sich allerdings herausgestellt, dass diese Situation nie auftreten kann. Da der Zuggenerator nur legale Züge ausgeben kann, kann zu einer Stellung, die matt ist, kein Folgezug berechnet werden. Somit stoppt die Zuggenerierung hier und die Engine sieht nie die Stellung, in der der König geschlagen und der Materialwert angepasst wurde. Wie wir die Erkennung von Matt stattdessen gelöst haben, steht in in Abschnitt 10.4

6.2 Architektur des Zuggenerators

In der Engine gibt es 8 Klassen, die zuständig sind für die Generierung von Folgezügen. Die Klasse `MoveGenerator` ist die Schnittstelle, über die `GameNode`- und `Position`-Objekte die Zuggenerierung ansteuern können. Die abstrakte Klasse `PieceMoveGenerator` ist das Muster für die Implementierung eines figurenspezifischen Zuggenerators und für jeden der 6 Figurentypen gibt es jeweils eine Klasse, die basierend auf `PieceMoveGenerator` die

Zuggenerierung für diesen Figurentyp übernimmt. Diese figurenspezifischen Zuggeneratoren bekommen als Eingabe jeweils die aktuelle Stellung, die Reihe und Linie der zu betrachtenden Figur und eine Ergebnisliste. Da sie so konstruiert sind, dass sie jeweils auf einem eigenen Thread ausgeführt werden können, gibt Java die Limitation vor, dass die Generatoren keine Rückgabetypen haben dürfen. Deshalb geben sie ihre Ergebnisse weiter, indem sie die übergebene Ergebnisliste in place verändern und alle berechneten Folgezüge an diese Listen anhängen.

Des Weiteren gibt es noch die Klasse `RowMoveGenerator`. Diese wird benötigt um eine nebenläufige Ausführung des Zuggenerators zu ermöglichen, bei der ein Thread jeweils mit der Berechnung der Folgezüge für eine Reihe des Spielfelds beauftragt wird.

6.3 Die einzelnen Figuren

6.3.1 Bauer

Bauern haben die Besonderheit, dass sie sich immer nur in eine einzige Richtung bewegen dürfen in Abhängigkeit von ihrer Farbe. Deswegen wird im `PawnMoveGenerator` zuerst überprüft, für welche Farbe die Folgezüge generiert werden müssen und im Zusammenhang damit ein Vorzeichen gesetzt, was in allen Hilfsfunktionen übergeben wird.

Bei jedem Aufruft des Zuggenerators für Bauern wird überprüft, ob für den aktuell konkret betrachteten Bauern ein Einzelzug, ein Doppelzug, ein Schlagen nach links, ein Schlagen nach rechts oder ein Schlagen en passant möglich ist. Jeder dieser Züge, der nach den Zugregeln des Bauern legal ist, wird einer Hilfsfunktion namens `addPawnMove()` übergeben.

Diese Hilfsfunktion prüft dann zuerst, ob der Zug den eigenen König im Schach zurücklassen würde. Sollte dies der Fall sein, dann bricht die Hilfsfunktion ab. Lässt der Zug den eigenen König nicht im Schach zurück, dann wird wird überprüft, ob der Zug zu einer Bauernumwandlung führt. Findet eine Bauernumwandlung statt, so wird eine Umwandlungsfunktion aufgerufen, die alle möglichen Umwandlungen in die Ergebnisliste schreibt.

Findet keine Bauernumwandlung statt, so wird stattdessen die Stellung, die aus dem Zug entsteht, der ursprünglich an `addPawnMove()` übergeben wurde, in die Ergebnisliste hinzugefügt. Eine Unterscheidung ob geschlagen wurde oder nicht ist hier nicht notwendig, da der Zähler für die 50-Züge-Regel in jedem Fall nach einem Bauernzug zurückgesetzt werden muss.

6.3.2 Springer

Der Zuggenerator für Springer hat 8 Züge, die er bei jedem Aufruf abprüft. Diese stimmen mit den 8 L-förmigen Bewegungsmöglichkeiten, die die Regeln für den Springer vorsehen, überein. Die Überprüfung läuft so ab, dass zuerst sichergestellt wird, dass der Zug das Spielfeld nicht verlassen würde. Ist er innerhalb der Grenzen des Spielfelds wird geschaut, ob das Feld, auf dem der Zug endet, besetzt ist. Ist es nicht besetzt, so wird der Zug der Hilfsfunktion `addKnightMove()` übergeben, die die aus einem Zug resultierende Stellung in die Ergebnisliste hinzufügt sofern der Zug den König nicht im Schach zurücklässt. Ist das Feld besetzt, so findet eine Fallunterscheidung statt: hat die Figur auf dem Feld dieselbe Farbe wie der ziehende Springer, so schlägt die Überprüfung fehl. Hat die Figur auf dem Feld allerdings nicht dieselbe Farbe wie der ziehende Springer, so wird der Zug an `addKnightMove()` übergeben und zusätzlich wird die Hilfsfunktion darüber informiert, dass eine Figur geschlagen wurde

Die Unterscheidung ob geschlagen wurde oder nicht ist notwendig, da nach 50 Zügen ohne dass ein Bauer gezogen hat oder eine Figur geschlagen wurde die Partie in einem Unentschieden endet.

6.3.3 König

Der Zuggenerator für Könige hat 10 Züge, die er bei jedem Aufruf abprüft, 8 reguläre und 2 Sonderzüge. Die regulären Züge sind das Betreten der 8 an das Startfeld grenzenden Felder und die Überprüfung dieser 8 Züge findet mit der Hilfsfunktion `addKingMove()` statt. Diese funktioniert analog zu `addKnightMove()` aus dem Abschnitt über den Springer-Zuggenerator. Die einzige Besonderheit, die `addKingMove()` hat, ist, dass die Funktion zusätzlich noch die Rochaderechte eines Königs in den Folgestellungen verwirft, nachdem dieser sich bewegt hat.

Die zwei Sonderzüge sind die beiden Rochaden, die ein König durchführen kann - die lange Rochade und die kurze Rochade.

Zur Überprüfung einer Rochade wird zuerst sichergestellt, dass der ziehende König sich gerade nicht im Schach befindet. Danach wird sichergestellt, dass das relevante Rochaderecht noch besteht (dieses hängt von der Farbe des ziehenden Königs ab und davon, ob eine lange oder kurze Rochade durchgeführt werden soll).

Ist das erledigt, dann überprüft der Zuggenerator, dass die zwei Felder, die der König passieren muss, beide frei und nicht im Schach sind. Bei einer kurzen Rochade sind das auch die einzigen Felder, die der Turm passieren muss, bei der langen Rochade muss der Turm noch ein zusätzliches Feld passieren, welches von einem Gegner angegriffen werden darf aber trotzdem frei (also nicht von einer Figur der eigenen oder anderen Farbe belegt) sein muss. Bei einer langen Rochade findet also zusätzlich eine Kontrolle statt, ob das Feld, das nur der Turm passieren muss, frei ist.

Schlägt eine der oben aufgezählten Kontrollschritte fehl, so wird die gesamte Überprüfung beendet und keine Rochade in die Ergebnisliste geschrieben. Sind alle Kontrollen erfolgreich, so wird der Rochadezug als Doppelschritt des Königs codiert und an die Hilfsfunktion `addKingMove()` weitergegeben, die sich genauso verhält wie bei den Einzelschritten des Königs, die vorher in diesem Abschnitt behandelt wurden.

6.3.4 Die Sliding Pieces

Als sliding Pieces bezeichnet man die Figuren, bei denen die Anzahl der möglichen zu ziehenden Felder nur von der Feldgröße und blockierenden Figuren begrenzt wird. Hierbei handelt es sich um die Dame, den Läufer und den Turm.

Alle 3 Zuggeneratoren für sliding Pieces benutzen eine Hilfsfunktion namens `computeRay()`. Diese Hilfsfunktion bekommt die aktuelle Stellung, die Koordinate der zu bewegenden Figur, eine horizontale Steigung und eine vertikale Steigung übergeben. Die beiden Steigungen bilden zusammen einen zweidimensionalen Vektor, der die Bewegung der Figur über das Feld beschreibt. Dann fügt `computeRay()` alle Züge, die ausgehend von der Startposition auf dem übergebenen Vektor liegen und den eigenen König nicht im Schach zurücklassen, zur Ergebnisliste hinzu. Wir dabei eine Figur geschlagen, so wird dies berücksichtigt und der Zähler für Halbzüge seit dem letzten Bauernzug oder Schlagen wird in Folgestellungen auf 0 gesetzt.

6.3.4.1 Läufer

`BishopMoveGenerator` nutzt eine Hilfsfunktion, die `computeRay()` viermal startet - einmal mit dem Vektor nach Nordosten ausgerichtet, einmal nach Südosten, einmal nach

Südwesten und einmal nach Nordwesten.

6.3.4.2 Turm

`RookMoveGenerator` nutzt ebenfalls eine Hilfsfunktion, die `computeRay()` viermal startet - einmal mit dem Vektor nach Norden ausgerichtet, einmal nach Osten, einmal nach Süden und einmal nach Westen.

Zusätzlich wird der Hilfsfunktion mitgeteilt, dass es sich hierbei um eine Bewegung des Turmes handelt, wodurch die Rochaderechte für diesen Turm in den Folgestellungen verloren gehen.

6.3.4.3 Dame

`QueenMoveGenerator` benutzt die Hilfsfunktion der anderen beiden sliding Pieces mit. Die Hilfsfunktion für den Läufer kann einfach genauso aufgerufen werden. Der Hilfsfunktion für den Turm wird allerdings mitgeteilt, dass es sich dieses Mal nicht um eine Bewegung des Turmes handelt, wodurch die Rochaderechte in den Folgestellungen unverändert bleiben.

6.4 Nebenläufigkeit beim Zuggenerator

Nach einigen Experimenten mit der Schachengine ist uns aufgefallen, dass die Generierung von Zügen ohne Nebenläufigkeit extrem viel Zeit kostet. Um die Performance der Engine zu steigern haben wir verschiedene Ansätze ausprobiert.

6.4.1 Aufteilung auf Threads

Der erste Ansatz, der umgesetzt wurde, war, dass die von außen ansprechbare Funktion in `MoveGenerator` einmal über das Spielbrett iteriert und auf jedem belegten Feld der ziehenden Farbe eine Instanz des figurenspezifischen Zuggenerators für den Figurentyp erstellt, der dieses Feld belegt. Diese Generatoren implementieren ein Java-Interface, wodurch die Instanzen jeweils einem eigenen Thread zugewiesen werden können. Ihr Ergebnis schreiben die instanziierten Zuggeneratoren dann in eine geteilte, threadsichere Liste.

Dieser Ansatz hat allerdings zu viel Overhead geführt, da in sehr schneller Frequenz neue Threads gestartet und alte Threads geschlossen wurden, sodass in Experimenten dieser Ansatz ungefähr dreimal so lange gerechnet hat wie eine Implementierung ohne Nebenläufigkeit.

Der zweite Ansatz, der dann implementiert wurde, war, statt für jedes Feld einen eigenen Thread zu eröffnen immer 8 Felder in einem Thread abzuarbeiten. Hier wurde der oben bereits kurz angesprochene `RowMoveGenerator` gebaut.

In diesem Ansatz instanziiert die Hilfsfunktion in `MoveGenerator` für jede Zeile einen eigenen `RowMoveGenerator`, der dann auf einem eigenen Thread über die ihm zugewiesene Zeile iteriert. Wichtig ist hierbei, dass die figurenspezifischen Zuggeneratoren nicht instanziiert werden, sondern dass die figurenspezifischen Generatoren statische Methoden zur Verfügung stellen, die der jeweilige `RowMoveGenerator` dann aufrufen kann.

Hierbei wurde der `Overhead` deutlich reduziert, ein nebenläufiger Zuggenerator mit dem zweiten Ansatz benötigte in Tests nur halb so lange wie ein nebenläufiger Zuggenerator mit dem ersten Ansatz. Er benötigte aber trotzdem noch eineinhalbmal so lange wie ein Zuggenerator ohne Nebenläufigkeit.

6.4.2 Speicherung der Ergebnisse

Der nächste Punkt, an dem wir optimiert haben, war die Speicherung der Ergebnisse. Statt die Ergebnisse alle in dieselbe threadsichere Liste zu schreiben haben wir für jede der 8 `RowMoveGenerator`-Instanzen eine eigene Liste angelegt und die Ergebnisse in diese Listen geschrieben. `MoveGenerator` hat dann mit dem `join()`-Befehl gewartet, bis alle `RowMoveGenerator`-Instanzen ihre Berechnungen abgeschlossen haben, dann die 8 einzelnen Ergebnislisten in eine Liste zusammengefasst und diese eine Liste zurückgegeben.

Dies hat in einigen Fällen starke Verbesserungen gebracht, aber im Großen und Ganzen lag die Implementierung immer noch hinter der Implementierung ohne Nebenläufigkeit zurück

6.4.3 Threads und Thread Pools

Der Durchbruch kam dann, als die Engine vom Arbeiten auf einzelnen Threads umgestellt wurde zum Arbeiten mit einem Thread Pool. Der Thread Pool hält immer eine feste Anzahl an Threads am Laufen. Soll nun eine Aufgabe nebenläufig ausgeführt werden, so muss nicht jedes Mal ein neuer Thread erstellt werden. Stattdessen wird die Aufgabe dem Thread Pool übergeben. Dieser lässt sie dann auf einem bereits existierenden, unbeschäftigten Thread laufen, wodurch der Overhead durch das ständige Erstellen und Löschen neuer Threads komplett eliminiert wird.

In Experimenten zeigte sich, dass die Implementierung mit Thread Pool ungefähr 40% schneller rechnete als die bisher schnellste Alternative, die Implementierung des Zuggenerators ohne Nebenläufigkeit.

7 Attack Maps

7.1 Warum Attack Maps?

Beim Schach gilt die Regel, dass keine Partei ihren Zug beenden darf, während ihr König im Schach steht. Die Engine darf natürlich nur legale Züge spielen, unabhängig davon, ob die Überprüfung auf die Legalität schon im Zuggenerator stattfindet oder erst nach der Erzeugung aller möglichen pseudolegalen Züge.

Diese beiden Umstände führen dazu, dass die Engine in der Lage sein muss, zu überprüfen, ob die Ausführung eines Zuges dazu führt, dass der eigene König danach im Schach steht. Wir haben uns entschlossen, Attack Maps zu nutzen um herauszufinden, welche Felder angegriffen werden und welche nicht.

Das Aufbauen der beiden Attack Maps für die beiden Spielfarben findet immer dann statt, wenn ein neues `Position`-Objekt instanziiert wird. Danach wird aus den Attack Maps für beide Seiten herausgelesen, ob der jeweilige König sich im Schach befindet und welche der Felder, die bei einer Rochade passiert werden müssen, gerade angegriffen werden. Die Ergebnisse werden dann im `Position`-Objekt gespeichert, die Attack Maps selbst werden verworfen um Platz zu sparen.

Gibt es für eine Stellung bereits eine Attack Map, dann ist die Überprüfung, ob einer der beiden Könige im Schach steht, nur noch ein Zugriff auf ein Feld der Attack Map und somit sehr günstig im Bezug auf die Laufzeit. Die teurere Komponente der Schachüberprüfung ist hierbei also das Erstellen der Attack Maps.

7.2 Wie funktionieren unsere Attack Maps?

Zum Erstellen der Attack Maps haben wir eine eigene Klasse namens `AttackMoveGenerator` geschrieben, deren öffentliche Methode als Eingabe eine Farbe und ein Spielfeld bekommt.

Sie iteriert über das übergebene Spielfeld und markiert auf einer Repräsentation des Spielbretts diejenigen Quadrate, die von einer Figur der angegebenen Farbe angegriffen werden. Dafür benutzt sie die Funktion `paintRayAttack()`. Diese hat als Parameter das Spielfeld, für das eine Attack Map erstellt werden soll, die unfertige Attack Map , eine Startkoordinate für den Angriff und wieder zwei Steigungswerte, die einen zweidimensionalen Vektor bilden.

Ist der Vektor der Nullvektor, so wird nur die übergebene Startkoordinate markiert. Dies wird für Bauer, König und Springer genutzt, da diese keine Sliding Pieces sind.

Für die Sliding Pieces Läufer, Turm und Dame werden jeweils Angriffsvektoren mit einer Länge > 0 übergeben und die unfertige Attack Map wird entlang dieser Vektoren markiert. Die Markierung erfolgt bis der Rand des Spielfeldes erreicht ist oder der Strahl mit einer Figur kollidiert.

Die Markierungen finden sowohl bei den Sliding Pieces als auch bei den Non-Sliding Pieces in place auf der übergebenen, unfertigen Attack Map statt.

7.3 Repräsentationen des Spielbretts

7.3.1 Boolean-Arrays

Unser erster Ansatz war, die Attack Maps jeweils als ein zweidimensionales Boolean-Array mit den Dimensionen 8×8 anzulegen.

Dies führte dazu, dass wir für jede Stellung zwei Attack Maps erzeugen; eine für die Felder, die von weißen Figuren angegriffen werden und eine für Felder, die von schwarzen Figuren angegriffen werden. Das Erzeugen von zwei Attack Maps für die zwei Farben ist ein Verhalten, was wir über die Zeit auch beibehalten haben. Enthält ein Feld in diesem Array den Wert `true`, so wird dieses Feld von mindestens einer Figur der Farbe, zu der die Attack Map gehört, angegriffen. Enthält ein Feld den Wert `false`, so wird es nicht angegriffen.

Im Laufe des Projekts haben wir, um Speicherplatz zu sparen und Performance zu gewinnen, das zweidimensionale 8×8-Array zu einem eindimensionalen Array mit demselben Fassungsvermögen umgebaut.

7.3.2 Byte-Arrays

Um noch mehr Speicherplatz zu sparen und die Performance des Programms noch weiter zu verbessern haben wir uns gegen Ende des Projekts dazu entschieden, die Attack Maps mit einem eindimensionalen Byte-Array der Länge 8 darzustellen.

Bei diesem Ansatz entspricht jedes Byte im Array einer Reihe des Spielbretts und jedes Bit in dem Byte einem Quadrat in dieser Reihe. Wird ein Quadrat von der zu der Attack Map gehörenden Farbe angegriffen, so wird das Bit, das dieses Quadrat codiert, auf 1 gesetzt. Bits, die Quadrate codieren, die nicht angegriffen werden, werden auf 0 gesetzt. Diese Implementierung nutzt `setBitToTrue()`, eine Funktion, die als Parameter eine Attack Map und eine Koordinate auf dieser Attack Map bekommt und in der übergebenen Attack Map das Bit, das die angegebene Stelle codiert, in place auf 1 setzt.

Außerdem stellt die Klasse `AttackMapGenerator` für Funktionen, die die Attack Maps lesen müssen, `getBoolFromByte()` zur Verfügung.

Diese Funktion hat die gleichen Parameter wie `setBitToTrue()` und gibt `true` aus, wenn der Wert des die Koordinate darstellenden Bits in der übergebenen Attack Map 1 ist und `false` wenn das Bit den Wert 0 hat.

7.4 Nebenläufigkeit im `AttackMapGenerator`

Bei Experimenten ist uns aufgefallen, dass viel Zeit bei der Berechnung des besten Zuges für das Erstellen von Attack Maps benötigt wird.

In dem Versuch, eine bessere Rechenzeit zu erreichen, haben wir den Thread Pool des `MoveGenerators` vergrößert und den `AttackMapGenerator` so umgebaut, dass er für jede Reihe, über die er iteriert, eine neue Aufgabe an den Thread Pool übermittelt. So wurde jedes Anfordern einer Attack Map auf jeweils 8 Threads verteilt.

Diese Veränderung führte allerdings nicht zu der erhofften Verbesserung; stattdessen brauchte die Engine bei gleicher Tiefe das vier- bis fünffache an Rechenzeit im Vergleich zur Variante ohne nebenläufigen `AttackMapGenerator`. Unsere Vermutung ist, dass durch die Kombination von Nebenläufigkeit im `MoveGenerator` und im `AttackMapGenerator` zu viele Threads gleichzeitig in Verwendung waren, was zu Blockierungen und langen Wartezeiten geführt hat.

Deshalb haben wir uns dazu entschlossen, die Veränderungen nicht in das fertige Produkt zu übernehmen.

7.5 Weitere Optimierungsansätze

Anstatt die Attack Maps jedes Mal von neu aufzubauen wäre eine Variante denkbar, bei der die Attack Map des vergangenen Zuges und der zuletzt gespielte Zug als `Move`-Objekt mit übergeben werden. In dieser Variante würden dann nur die Veränderungen des letzten Zuges auf einer bestehenden Attack Map ausgeführt werden.

Aus Zeitknappheit waren wir allerdings nicht mehr in der Lage, diesen Ansatz auszuprobieren um zu überprüfen, ob die Veränderungen tatsächlich zu einer höheren Effizienz der Engine geführt hätten.

8 Erkennung von dreifacher Stellungswiederholung

8.1 Warum ist eine Erkennung von Stellungswiederholungen notwendig?

Tritt in einer Schachpartie dieselbe Stellung dreimal auf, dann kann die Partei, die gerade am Zug ist, diese Partie für unentschieden erklären lassen. Automatisch unentschieden ist eine Partie nach den FIDE-Regeln erst, nachdem dieselbe Stellung fünfmal auftritt. Da das UCI-Protokoll allerdings keine Möglichkeit gibt, einen Antrag auf Unentschieden zu stellen, erklärt die Benutzeroberfläche Arena ein Spiel für unentschieden, wenn eine UCI-Engine am Zug ist, während dieselbe Stellung das dritte Mal auftritt. Deswegen haben wir uns dazu entschieden, in unserer Engine das dreifache Auftreten derselben Stellung auch als ein automatisches Unentschieden zu werten, unabhängig davon, ob die Partei,

die gerade am Zug ist, im Vorteil ist und vielleicht die Partie weiterführen würde, wenn sie die Wahl hätte.

Zwei Stellungen sind identisch im Sinne der Wiederholungsregel genau dann, wenn

- die gleichen Figurentypen auf den gleichen Feldern stehen
- dieselbe Partei am Zug ist
- die Rochaderechte gleich sind
- die Möglichkeit eines En-Passant-Schlagens die gleiche ist

.

8.2 Speicherung der bisher ausgeführten Stellungen

Die Erkennung einer dreifachen Wiederholung setzt voraus, dass eine Historie geführt wird, in der alle bisher gespielten Stellungen erfasst sind. Zur Realisierung dieser Historie haben wir verschiedene Ansätze ausprobiert.

8.2.1 In den `Position`-Objekten

Der erste Ansatz, den wir verfolgt haben, war der, dass jedes `Position`-Objekt eine Liste mit allen vergangenen Stellungen speichert. Dieser Ansatz war einfach zu implementieren, hat allerdings zu stark erhöhtem Speicherbrauch und einer dreimal so langen Laufzeit des Programmes geführt.

8.2.2 In einer öffentlichen, statischen Liste

Der Ansatz, den wir in der fertigen Engine benutzen ist der, eine zentralisierte Liste zu führen mit allen bisher gespielten Stellungen. Die Stellungen sind kodiert in Form von Strings, die die oben genannten, für die Wiederholungsregel relevanten Attribute wiedergeben. Diese Liste ist eine öffentliche, statische Klassenvariable, auf die von überall aus zugegriffen werden kann. Wird eine Stellung gespielt, so wird eine String-Repräsentation von ihr in diese statische Liste eingefügt. Wird eine Stellung während der berechnung des optimalen Zuges betrachtet, ohne jedoch bisher tatsächlich gespielt worden zu sein - also als eine mögliche Stellung in der Zukunft - dann wird diese Stellung in die statische Liste eingefügt bevor ihre Folgestellungen ausgewertet werden und nach der Auswertung der Folgestellungen wieder aus der Liste entfernt. So wird die Erkennung von dreifacher Stellungswiederholung möglich, auch wenn unter den bisher tatsächlich gespielten Stellungen noch nicht eine einzige doppelt vorgekommen ist.

Die Performance wurde durch die Änderung stark verbessert, leider bedeutete das Implementieren der Erkennung von dreifacher Stellungsbewertung trotzdem eine deutliche Verlangsamung von bis zu 10 Sekunden. Da die Engine pro Zug 30 Sekunden Zeit zum Rechnen hat kann es also sein, dass ein Drittel der Rechenzeit darauf verwendet wird, zu erkennen, ob es zu einem Unentschieden gekommen ist oder nicht.

8.2.3 In einer HashMap

Da der Ansatz, der die öffentliche, statische Liste nutzt, immer noch mit einer merklichen Verschlechterung der Performance verbunden war, war dieser nicht der letzte Ansatz, den wir ausprobiert haben. Da bei der Überprüfung, ob eine Stellung bereits mindestens

dreimal gespielt wurde, alle bisher ausgeführten Stellungen in eine HashMap geschrieben werden (das genaue Verfahren ist nachzulesen im nächsten Abschnitt) hatten wir gehofft, einen Performance-Gewinn zu erzielen, indem wir die Historie direkt in einer HashMap führen.

Bei diesem Ansatz ergaben sich allerdings zwei Probleme: zuerst einmal gab es ein Platzproblem - die Java-Garbage-Collection hat viele der geleerten Felder der HashMap nicht weggeräumt, wodurch der Speicherverbrauch um das Fünffache gestiegen ist. Wir haben versucht, diesem Problem entgegenzuwirken, indem wir eine Variante der HashMap namens WeakHashMap genutzt haben, die stärker mit der Garbage Collection zusammenarbeitet. Diese ist aber nur sehr schlecht geeignet, um Daten über einen längeren Zeitraum zu persistieren, da teileweise die Garbage Collection zu früh eingegriffen hat.

Das zweite Problem war, dass die Berechnung länger gedauert hat als vorher. Die ständigen Schreib- und Lösch-Zugriffe auf die statische HashMap (vor Allem getätigt von den Algorithmen zur Berechnung des besten Zuges, die häufig Stellungen in die Historie schreiben und sie wieder herauslöschen) haben deutlich länger gebraucht als dieselben Zugriffe auf die öffentliche, statische Liste.

Aus diesem Grund haben wir uns entschlossen, die Änderungen nicht in die finale Version der Engine einzubauen und stattdessen eine separate HashMap zu benutzen für das Nachzählen der Stellungen in der Historie.

8.3 Überprüfung, ob eine Stellung dreimal in der Liste vorkommt

Um zu überprüfen, ob eine Stellung dreimal in der Liste aller bisher gespielten Stellungen vorkommt, existiert eine private, statische String-Integer-HashMap in der Klasse `Position`. Eine HashMap ist eine Datenstruktur, die mit Schlüssel-Wert-Paaren funktioniert. In unserem Fall sind die Schlüssel Strings und die Werte Integer. Man gibt der `get()`-Methode der HashMap also einen String als Argument und erhält als Rückgabe einen Integer-Wert. Jedem String-Schlüssel ist genau ein Feld in der HashMap zugeordnet das genau einen Integer-Wert enthält.

Die statische HashMap in `Position` ist zugänglich für jede Instanz der Klasse. Sie wird immer wieder verwendet und geleert, bevor die Historie auf Wiederholungen geprüft wird, damit nicht für jede Überprüfung eine neue HashMap instanziiert werden muss. So wird vermieden, dass immer neuer Speicher allokiert und wieder freigegeben wird.

Das Verfahren befindet sich in der Funktion `checkForThreefoldRepetition()` in der Klasse `Position`. Es besteht darin, über die Historie mit den Stellungsstrings zu iterieren und für jeden String in der statischen HashMap in `Position` nachzuschauen, ob es bereits einen Eintrag für diesen String gibt. Gibt es noch keinen, so wird ein neuer Eintrag erstellt und mit 1 initialisiert, um festzuhalten, dass dieser String bereits einmal in der Liste vorgekommen ist. Gibt es aber bereits einen Eintrag für diesen Schlüssel, so wird der Wert des Inhalts dieses Eintrags um 1 erhöht, um zu reflektieren, dass ein weiteres Vorkommnis des Schlüssel-Strings gefunden wurde. Dann wird geschaut, ob der Wert des Inhalts des gerade betrachteten Feldes größer oder gleich der Zahl 3 ist. Sollte dem so sein, dann wurde eine Stellung gefunden, die mindestens dreimal gespielt wurde und `checkForThreefoldRepetition()` gibt `true` zurück. Sollte dem nicht so sein am Ende der Schleife, dann gibt `checkForThreefoldRepetition()` den Wert `false` zurück.

9 Spielbaum

Der Spielbaum ist die zentrale Datenstruktur der Engine. Es handelt sich hierbei um einen Baum, in dem jeder Knoten eine Stellung speichert. Kinder eines Knotens enthalten eine Stellung, die aus der Stellung des Elternknotens in einem Zug erreichbar ist. Es muss sich hierbei um einen Zug der aktiven Partei handeln, welcher in jeder Ebene wechselt. Somit stellt jeder Pfad eine legale Zugfolge dar.

Der Spielbaum wird zu einer gegebenen Stellung berechnet. Hierzu wird diese Stellung als Wurzel des Baums verwendet. Dann werden alle in der Stellung für der aktiven Partei legalen Züge auf die Stellung angewandt, und die resultierenden Stellungen als Kinder an die Wurzel angehängt. Wiederholt man dies rekursiv für alle Kindknoten, so ergibt sich ein Baum, der alle Möglichkeiten des weiteren Spielverlaufs enthält. In der Praxis muss man den Spielbaum auf eine maximale Tiefe begrenzen.

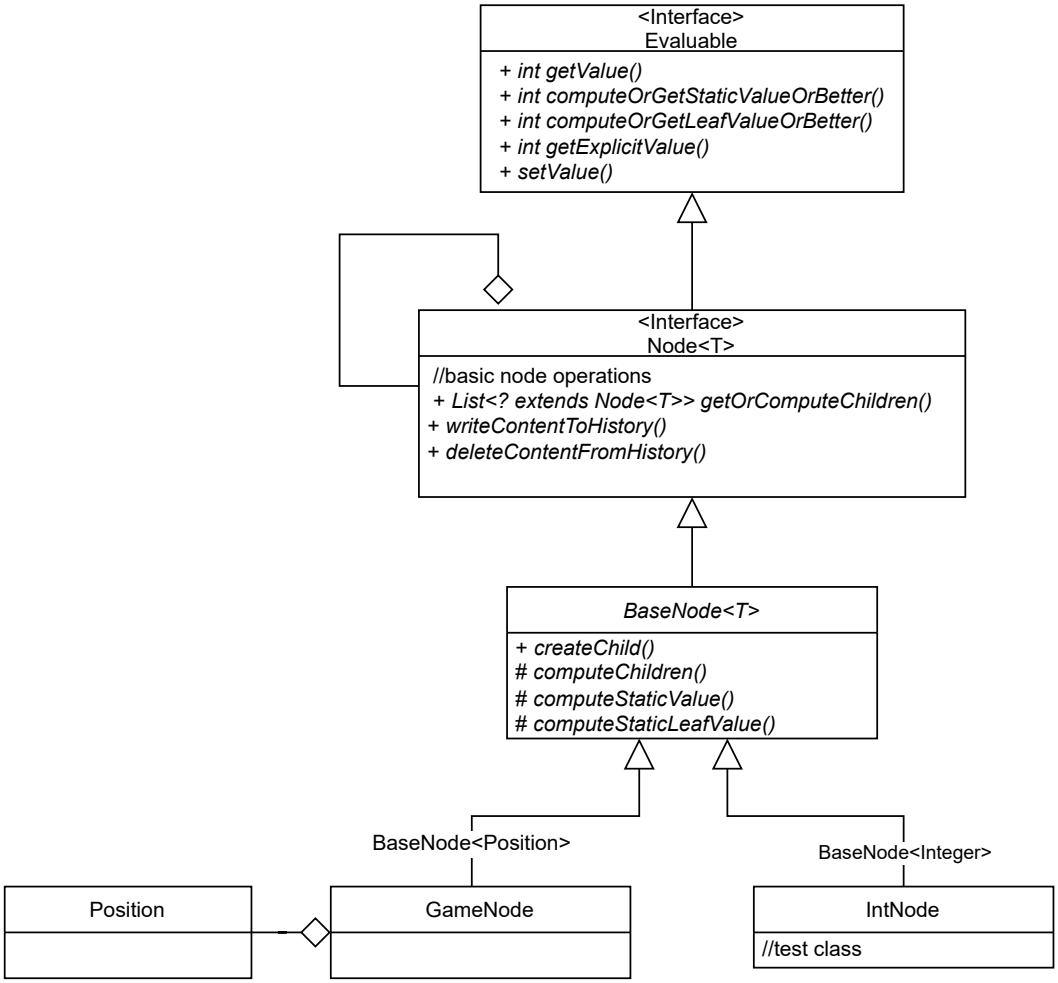

Abbildung 9.1: Klassendiagramm des Spielbaums

9.1 Interface `Node`

Der Spielbaum setzt sich aus Knoten zusammen, die das Interface `Node` implementieren. Hierbei handelt es sich um ein generisches Interface, das gängige Knotenmethoden deklariert. Zusätzlich deklariert das Interface die Methoden `writeContentToHistory` und

`deleteContentFromHistory`, welche zur Erkennung von Stellungswiederholungen genutzt werden. Siehe hierzu Kapitel 8.

Die Methode `getOrComputeChildren()` wird benutzt um auf die Kinder eines Knotens zuzugreifen. Hat ein Knoten keine Kinder, so wird versucht diese zu berechnen. Können keine Kinder berechnet werden, so wird eine `ComputeChildrenException` ausgelöst. Damit gibt diese Methode nie `null` zurück und kann verwendet werden, ohne vorher zu überprüfen, ob der Knoten überhaupt Kinder besitzt.
Die Methode verbirgt die Berechnung von Kindern und wird benutzt, um Algorithmen zu schreiben, welche sowohl auf bereits vollständig oder teilweise konstruierten Bäumen ausführbar sind, diese aber auch selbstständig aus einer Wurzel berechnen können.

9.2 Interface `Evaluable`

Das Interface `Evaluable` handhabt die Auswertung von Knoten. Es wurde entwickelt, um unter Iterative Deepening Informationen aus früheren Auswertungen nutzen zu können. Hierzu wird ein drei-stufiges System verwendet.

1. Explizit zugewiesene Bewertung
2. Bewertung von Blättern
3. statische Bewertung

Ein `Evaluable` kann zu jeder Zeit nur einen Wert haben. Die Bewertungen überschreiben einander in der gegebenen Reihenfolge. In der Aufzählung weiter oben stehende Bewertungen werden als höherstufig bezeichnet und können weiter unten stehende Bewertungen überschreiben. Eine statische Bewertung kann durch eine Blattbewertung oder einen explizit zugewiesenen Wert überschrieben werden, eine Blattbewertung kann nur durch das explizite Zuweisen einer Bewertung überschrieben werden. Ein explizit zugewiesener Wert kann jederzeit durch eine neue explizite Zuweisung überschrieben werden.

Um einen Knoten auszuwerten, wird `computeOrGetStaticValueOrBetter()` oder `computeOrGetLeafValueOrBetter()` aufgerufen. Beide Methoden dürfen die geforderte Bewertung oder eine höherstufige Bewertung zurückgeben, falls diese bereits gespeichert ist. Ist nicht mindestens der geforderte Wert gespeichert, so wird er berechnet und für zukünftige Aufufe gespeichert.

Um einem Knoten eine explizite Bewertung zuzuweisen, kann die Methode `setValue()` genutzt werden. Die Methode `getExplicitValue()` ermöglicht den Zugriff auf solch einen Wert. Wurde keine Bewertung explizit zugewiesen, wird eine Ausnahme ausgelöst.

Die Methode `getValue()` gibt den aktuell gespeicherten Wert zurück. Hierbei handelt es sich um die höchststufige bereits bestimmte Bewertung. Wurde noch kein Wert bestimmt oder zugewiesen, wird eine Ausnahme ausgelöst.
Da `computeOrGetStaticValueOrBetter()` von den beiden anderen Bewertungen überschrieben wird, stellt es eine Alternative zu `getValue()` dar, welche bei fehlender Bewertung keine Ausnahme auslöst, sondern eine statische Auswertung vornimmt.

9.3 BaseNode

Die abstrakte Klasse `BaseNode` stellt eine grundlegende Implementierung der `Node` und `Evaluable` Interfaces zur Verfügung. `BaseNode` wurde so entwickelt, dass sich ganzzahlige Bäume und Spielbäume möglichst viel Code teilen. Ziel war es, dass statt der Produktivimplementierung `GameNode` möglichst oft eine von `BaseNode` abgeleitete Testklasse wie `IntNode` eingesetzt werden kann, um den Testaufwand zu verringern.

Hierzu wurden die Klasse selbst und das Interface `Node` generisch implementiert. Dies ermöglich es Subklassen, beliebige Datentypen in Knoten zu speichern - beispielsweise Schachstellungen (`Position`) oder ganze Zahlen.

Weiterhin wurden nicht vollständig implementierbare Methoden soweit wie möglich implementiert und durch abstrakte Methoden unterstützt. Nicht vollständig implementiert sind unter anderem `getOrComputeChildren()`, `computeOrGetStaticValueOrBetter()` und `computeOrGetLeafValueOrBetter()`. Die von diesen Methoden benötigten Speichermechanismen werden durch `BaseNode` implementiert und rufen abstrakte Methoden zur tatsächlichen Berechnung von Kindknoten oder Bewertung auf. Diese müssen durch Subklassen implementiert werden, da nur diese Kenntnis über den Typ der gespeicherten Daten haben.

10 Auswertung

Soll die Engine einen Zug spielen, so werden ausgehend von einer gegebenen Stellung die möglichen Folgezüge bewertet. Der als am besten bewertete Zug wird dann von der Engine gespielt. Die Stärke der Engine hängt maßgeblich von den implementierten Auswertungsmechanismen ab. Daher wurden diese mehrfach erweitert, um die Spielstärke der Engine zu erhöhen.

Im Kontext der Auswertung stellt eine positive Bewertung immer einen Vorteil für die weißen Seite dar, negative Werte einen Vorteil für den schwarzen Seite.

10.1 Minimax und verbesserte Varianten

Zur Auswertung des Spielbaums und Bestimmung des nächsten Zugs setzt die Engine den Minimax-Algorithmus beziehungsweise verbesserte Weiterentwicklungen davon ein. Da die Weiterentwicklungen stets das gleiche Ergebnis wie Minimax ermitteln, und lediglich effizienter arbeiten, werden diese im Folgenden als Varianten von Minimax bezeichnet.

Jede Variante stellt eine eigene Klasse dar und befindet sich im `minimax`-Paket.

Um gemeinsam mit `BaseNode` aussagekräftige Tests mit ganzen Zahlen statt Schachstellungen zu ermöglichen, sind auch alle Minimax-Varianten generische Klassen.

10.1.1 Minimax

Minimax ist implementiert in der Klasse `GenericMiniMax`.

Der Minimax-Algorithmus ermittelt den bestmöglichen Zug und bildet die Grundlage der Baumauswertung. Da ein Spielbaum für Schach einen hohen Verzweigungsgrad

aufweist, kann ausgehend von einer Stellung nicht der gesamte restliche Spielverlauf berechnet werden. Man muss sich stattdessen auf eine Anzahl Halbzüge beschränken, die man vorrausschaut. Die Anzahl der Züge bestimmt die Tiefe des Spielbaumes, da jeder Knoten für einen Zug steht.

Der Spielbaum wird bis zu einer gewissen Tiefe aufgebaut und den Blattknoten werden statisch berechnete Werte zugewiesen. Zum Vorgehen der statischen Bewertung siehe Abschnitt 10.3.

Beide Parteien wechseln sich ab und spielen stets den für sie besten Zug. Der beste Zug ist hierbei der Zug, der am Ende der vorrausgeschauten Züge zur besten Stellung führt, falls der Gegner stets den für ihn besten Zug macht. Wichtig ist hierbei also, dass nicht stumpf der Zug ausgewählt wird, der auf dem Pfad zur besten Stellung liegt, sondern der Zug, von dem aus einem der Gegner am wenigsten schaden kann.

Da positive Bewertungen einen Vorteil für Weiß und negative Bewertungen einen Vorteil für Schwarz bedeuten, spielt Weiß demnach maximierend, schwarz minimierend. Da nach jedem Zug die aktive Partei wechselt, wird abwechselnd maximiert und minimiert (daher der Name). Nach diesem Vorgehen werden ausgehend von den Blattwerten die Werte der inneren Knoten bestimmt. Der Wert jedes Knotens entspricht dabei dem Wert seines besten Kindes - da dies der Zug ist, den die aktive Partei an diesem Knoten spielen wird. Der Zug, der den Wert der Wurzel bestimmt, ist der für die aktive Partei beste Zug. Siehe Abbildung 10.1 für eine grafische Darstellung.

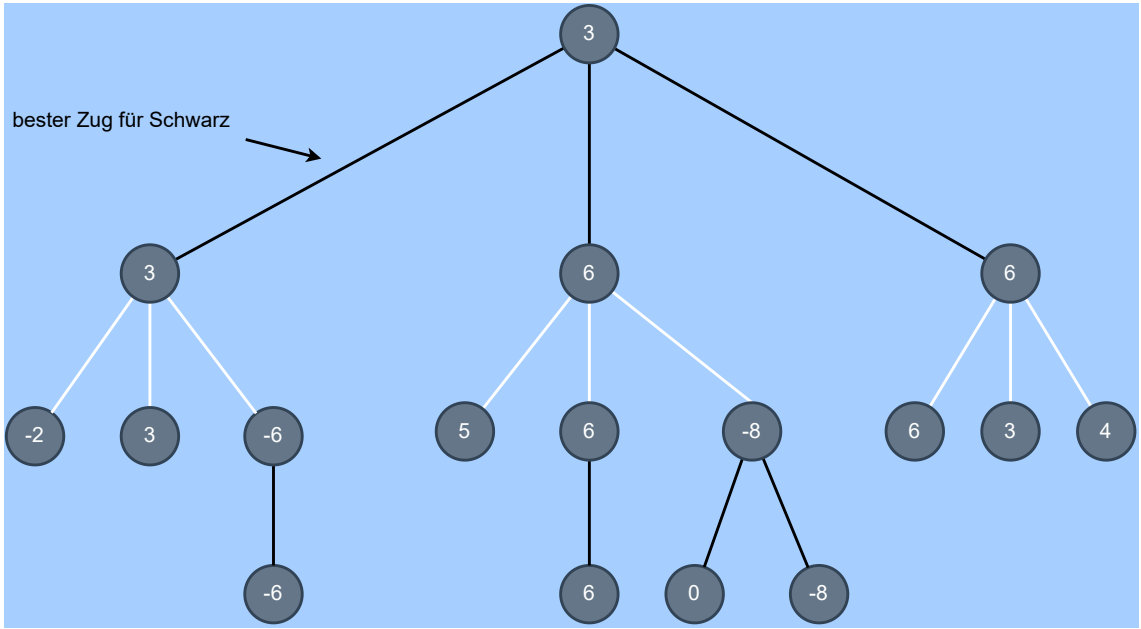

Abbildung 10.1: Minimax
eigenes Werk

10.1.2 Alpha-Beta-Pruning

Alpha-Beta-Pruning ist implementiert in der Klasse `GenericAlphaBetaPruning`.

Alpha-Beta-Pruning ist eine verbesserte Variante von Minimax. Während Minimax stets alle Knoten des Spielbaums betrachtet, können mit Alpha-Beta-Pruning Knoten ausgelassen werden. Alpha-Beta-Pruning garantiert hierbei das gleiche Ergebnis wie Minimax.

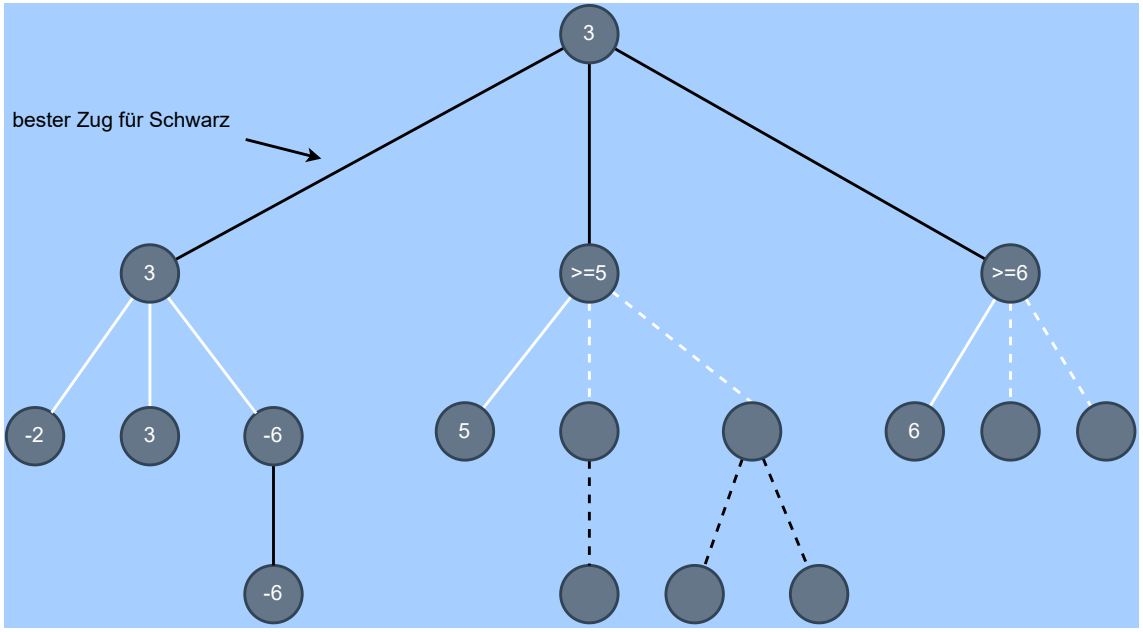

Abbildung 10.2: Alpha-Beta-Pruning
eigenes Werk

Abbildung 10.2 dient als Beispiel. Der linke der drei Teilbäume wird vollständig ausgewertet. Bei der Auswertung des mittleren Teilbaumes ergibt sich für den ersten Blattknoten eine statische Bewertung von 5. Hier ist weiß am Zug. In der oberen Ebene ist schwarz am Zug. Schwarz kann den Zug spielen, der zum ersten Teilbaum führt, und erhält mindestens eine Stellung mit Bewertung 3. Alle Alternativen in diesem Teilbaum führen zu einer noch niedrigeren Bewertung, welche noch besser für Schwarz ist. Spielt Schwarz den Zug, der zum mittleren Teilbaum führt, so kann Weiß mindestens eine Bewertung von 5 erzielen. Dies ist schlechter für Schwarz als die im linken Teilbaum garantierte 3. Unabhängig davon, wie niedrig oder hoch die restlichen Werte im mittleren Teilbaum sind, kann Weiß dort eine Bewertung von 5 oder höher erzielen. Also wird Schwarz diesen Teilbaum nicht spielen. Die restlichen Knoten des Teilbaums müssen deshalb nicht ausgewertet werden. Analog kann im rechten Teilbaum nach Auswertung des ersten Knotens mit 6 abgebrochen werden, da Schwarz diesen Baum nicht spielen wird.

10.1.3 Speicherverbrauch reduzieren

Der Spielbaum wird während der Auswertung aufgebaut. Die ursprüngliche Alpha-Beta-Implementierung wies einen sehr hohen Speicherverbrauch auf. Dieses Problem wurde neben den in Kapitel 5 aufgeführten Optimierungen dadurch gelöst, dass Knoten direkt nach ihrer Auswertung wieder aus dem Spielbaum gelöscht werden.

Der Spielbaum wird also während der Auswertung aufgebaut und gleichzeitig wieder gelöscht. So befinden sich zu jeder Zeit nur wenige Knoten im Hauptspeicher.

Diese Optimierung ist implementiert in der Klasse `SelfDestructingAlphaBetaPruning`.

10.1.4 Move Ordering

Um die Performance weiter zu erhöhen, wurde Move Ordering implementiert. Hierbei handelt es sich um eine Verbesserung von Alpha-Beta-Pruning, um mehr Knoten ignorieren zu können. Diese Optimierung ist implementiert in der Klasse

`MoveOrderingSelfDestructingAlphaBetaPruning`.

Alpha-Beta-Pruning kann dann Teilbäume ignorieren, wenn ein Knoten so gut ist, dass die andere Partei den Teilbaum nicht spielen wird.

Sortiert man alle Knoten so, dass die für die aktive Partei bestbewerteten Knoten vorne stehen, kann Alpha-Beta-Pruning möglichst viele Knoten ignorieren. Da zu diesem Zeitpunkt die Bewertung der Knoten noch nicht bekannt ist, muss auf eine statische Bewertung als Indikator gesetzt werden.

Will man einen Knoten auswerten, so sortiert man zunächst seine Kinder nach deren statischer Bewertung, und wertet sie dann der Reihe nach aus. Diese Anpassung hat die Performance der Engine signifikant verbessert.

10.2 Iterative Deepening

Iterative Deepening ist ein Verfahren zum Zeitmanagement bei der Spielbaumauswertung, das auch eingesetzt werden kann, um die Performance weiter zu erhöhen. Obwohl wir versucht haben, beide Aspekte zu implementieren, konnten wir keine Leistungszuwächse durch Iterative Deepening erzielen.

10.2.1 Zeitmanagement mittels Iterative Deepening

Statt zu Beginn der Auswertung eine Zieltiefe festzulegen, führt Iterative Deepening mehrere Auswertungen in wachsender Tiefe aus. Man startet mit einer Auswertung in minimaler Tiefe und erhöht nach Beendigung jeder Auswertung die Tiefe um eins.

Dieses Verfahren ermöglicht es, bereits nach kurzer Zeit ein vorläufiges Ergebnis aus einer niedrigen Tiefe zu haben. Läuft die Zugzeit der Engine ab, so wird die aktuelle Auswertung abgebrochen und das Ergebnis der letzten abgeschlossen Auswertung verwendet.

10.2.2 Iterative Deepening als Optimierung

Iterative Deepening kann zu einer Verbesserung des Move Orderings, und damit zu einem Leistungszuwachs führen. Hierzu wird statt nach einer statischen Bewertung der Knoten zu sortieren, auf Informationen aus vorherigen Auswertungen zurückgegriffen.

Um dies zu ermöglichen, wurde die Klasse `Evaluable` mit einem mehrstufigen Wertesystem entworfen. So kann stets nach der genauesten bekannten Bewertung sortiert werden.

Zusätzlich musste die Löschung von Knoten aus dem Baum eingeschränkt werden, damit Werte aus früheren Auswertungen verwendet werden können. Hierzu wurde die Klasse `StoringMoveOrderingSelfDestructingAlphaBetaPruning` geschrieben, welche eine konfigurierbare Anzahl an Baumebenen beibehält. Wird beispielsweise die erste Kindebene beibehalten, so kann in folgenden Auswertungen mit höherer Tiefe auf die davor ermittelte Bewertung der Knoten zugegriffen werden, um diese zu sortieren. Diese Sortierung stellt einen besseren Indikator als die sonst zur Sortierung verwendete statische Bewertung dar.

Obwohl das Beibehalten von und Sortieren nach ermittelten Bewertungen getestet wurde, konnte keine allgemeine Leistungssteigerungen festgestellt werden. Aus Zeitmangel war es uns nicht möglich, eine ausreichend große Menge an Stellungen zu testen. Daher haben wir uns dafür entschieden, Iterative Deepening nur als Zeitmanagementstrategie, nicht aber zusammen mit `StoringMoveOrderingSelfDestructingAlphaBetaPruning` als Optimierung einzusetzen.

10.3 Statische Auswertung

Grundlage der Zugbewertung bildet die statische Bewertung einzelner Stellungen. Hierzu werden Materialwert und Piece-Square-Tables eingesetzt.

10.3.1 Materialwert

Eine Bewertung nach Material gibt der Engine ein rudimentäres Verständnis dafür, welche Partei in einer gegebenen Stellung im Vorteil ist. Die Figurenwerte basieren auf Larry Evans' Bewertungen, wobei wir die Dame etwas niedriger bewerten. Die ursprüngliche Bewertung ist aufrufbar unter `https://chess.fandom.com/wiki/Value#Larry_Evans%27_valuation`. Hierzu werden Figuren in Centipawns gemäß folgender Aufzählung bewertet:

- Bauer: 100
- Springer: 350
- Läufer: 375
- Turm: 500
- Dame: 900
- König: 20000

Um eine Stellung zu bewerten, werden die Werte aller weißen Figuren auf dem Brett addiert und die Werte aller schwarzen Figuren davon subtrahiert. Hat beispielsweise Weiß ausgehend von der Startstellung einen Turm verloren, so ergibt sich eine Bewertung von -500, da Schwarz einen Turm mehr besitzt. Der resultierende Wert stellt somit eine Bewertung des Bretts nach Material dar, wobei positive Werte gut für Weiß und negative Werte gut für Schwarz sind.

Zusätzlich erhält eine Partei, der zwei (oder mehr) Läufer kontrolliert, einen Bonus von 150 Punkten.

10.3.2 Piece-Square-Tables

Die Bewertung nach Materialwert führte zu zahlreichen gleich bewerteten Stellungen, da sich die Bewertung nur beim Schlagen einer Figur ändert. Mit dieser Bewertung vernachlässigte die Engine die Entwicklung ihrer Figuren und übernahm keinerlei Kontrolle über das Feld.

Insbesondere wurde sehr häufig der Zug `Ta8-b8` gespielt, falls der entsprechende Springer bereits bewegt wurde. Nach der Bewertung aller Folgezüge spielte die Engine den ersten Zug mit der besten Bewertung. Fand die Engine keinen Zug, um innerhalb der vorrausgeschauten Züge eine gegnerische Figur zu schlagen, so spielte sie (falls möglich) einen neutralen Zug, mit dem sie zumindest keine Figur verlor. Da der Zug `Ta8-b8` einer der am frühesten betrachteten Züge war, und er zu Beginn einer Partie selten die Bewertung veränderte, wurde er überdurchschnittlich oft gespielt. Um besser zu spielen, ist es nötig, dass die Engine einen Vorteil darin sieht, ihre Figuren zu entwickeln und mehr Felder zu kontrollieren.

Um der Engine ein Verständnis von Feldkontrolle zu vermitteln, wurden Piece-Square-Tables verwendet. Die Tabellen wurden von Tomasz Michniewski entworfen und dem

Chess Programming Wiki entnommen. Sie sind unter folgender URL aufrufbar: `www.chessprogramming.org/Simplified_Evaluation_Function#Piece-Square_Tables`.

Piece-Square-Tables erweitern die beschriebene Materialbewertung um Figur- und Feldspezifische Boni. Hierzu gibt es zu jeder Figur in jeder Farbe eine 8×8 Tabelle, die einer Figur auf jedem Feld einen Bonus oder Malus zuweist.

Als Beispiel diene die Tabelle für weiße Bauern.

```
    0,  0,  0,  0,  0,  0,  0,  0,  // 8
    50, 50, 50, 50, 50, 50, 50, 50, // 7
    10, 10, 20, 30, 30, 20, 10, 10, // 6
    5,  5, 10, 25, 25, 10,  5,  5,  // 5
    0,  0,  0, 20, 20,  0,  0,  0,  // 4
    5, -5,-10,  0,  0,-10, -5,  5,  // 3
    5, 10, 10,-20,-20, 10, 10,  5,  // 2
    0,  0,  0,  0,  0,  0,  0,  0   // 1
  //a   b   c   d   e   f   g   h
```

Ein Bauer auf `d2` ist statt 100 nur noch 80 Centipawns wert, da das Feld einen Wert von -20 hat. Wird der Bauer auf `d3` vorgerückt, wird ihm ein Wert von 100 zugewiesen, da das Feld einen Wert von 0 hat. Ein Doppelschritt auf `d4` führt sogar zu einer Bewertung von 120, da das Feld einen Wert von 20 hat. Hierdurch wird die Engine unter anderem dazu angereizt mit ihren Bauern die Mitte zu kontrollieren.

Um die Tabellen möglich effizient zu implementieren, wurden statt 8×8 Arrays eindimensionale Arrays der Länge 64 eingesetzt. Weiterhin existiert je Figur nur eine Tabelle für beide Farben. Um Werte für die schwarze Seite zu ermitteln, werden die Zugriffe horizontal gespiegelt und die gelesenen Werte mit -1 multipliziert.

10.4 Auswertung natürlicher Blätter

Ein natürliches Blatt ist ein Blatt im Spielbaum, das sich nicht in maximaler Tiefe befindet. Ein natürliches Blatt entsteht nur, wenn zu einem Knoten keine Kinder berechnet werden können.

Befindet sich ein Knoten hingegen in maximaler Tiefe, so wird er ein Blatt, ohne dass überprüft wird, ob Kinder zu ihm bestimmt werden können. Hierbei handelt es sich um Knoten, zu denen aus zeitlichen Gründen lediglich keine weiteren Folgezüge mehr betrachtet werden.

Stellungen, die in natürlichen Blättern gespeichert sind, sind immer Matt, Patt oder Remis, da nur in diesen Stellungen keine Züge mehr möglich sind. Derartige Stellungen werden durch `PositionEvaluator.evaluateLeafPosition()` bewertet. Patt und Remis werden hierbei mit 0 bewertet. Ein Matt wird mit Tiefe der Stellung im Spielbaum mal Königswert (20.000) bewertet, und falls weiß Matt ist zusätzlich mit -1 multipliziert. Das Multiplizieren mit der Tiefe sorgt dafür, dass ein früheres Matt einem Späteren vorgezogen wird.

11 Testsuite

11.1 Notwendigkeit einer Testsuite

Damit die Engine in der Lage ist ein korrektes Spiel zu absolvieren, müssen sämtliche Komponenten korrekt arbeiten und ineinandergreifen. Es muss sichergestellt sein, dass Zuggenerierung, Attack Maps, statische Stellungsbewertung, Spielbaum und Zugbewertung korrekt implementiert sind. Dies stellt einen erheblichen Testaufwand da, welchen wir durch die Verwendung zahlreicher Hilfsklassen und -methoden auf ein Minimum zu reduzieren versuchten. Dennoch ist der Testcode mit ungefähr 7000 Zeilen fast so umfangreich wie das etwa 8000 Zeilen umfassende Hauptprogramm.

Zusätzlich stellte die Wartung mehrer Minimax-Varianten eine weitere Herausforderung dar. Hier galt es eine Implementierung zu finden, mit der Tests einmalig geschrieben und dann auf alle Varianten angewandt werden können.

11.2 Aufbau der Testsuite

Die Testsuite setzt sich aus 4 Paketen zusammen:

- `classes` - Subklassen existierender Produktiv-Klassen zu Testzwecken
- `data` - Testdaten (z.B. Testbäume für Minimax)
- `helper` - Klassen mit Hilfsmethoden
- `tests` - Klassen mit den tatsächlichen Tests

11.3 Paket `classes`

Das Paket `classes` enthält mehrere von `BaseNode` abgeleitete `Node`-Implementierungen, eine Ausnahme, die zum Abbrechen von Tests verwendet werden kann und einen `Comparator` zum Sortieren von `Position`-Objekten. Da der Komparator letzendlich nicht eingesetzt wurde, wird im Folgenden nicht weiter auf ihn eingegangen.

11.3.1 `FailureException`

Die Ausnahme `FailureException` wird zum Tunneln von Checked Exceptions in Tests genutzt. Checked Exceptions sind eine Gruppe von Ausnahmen, deren mögliches Auftreten entweder behandelt oder als Teil der Methodensignatur deklariert werden muss. Wird die Ausnahme in die Signatur einer Methode aufgenommen, müssen auch alle anderen Methoden, die diese Methode aufrufen, die Ausnahme deklarieren oder behandeln.

Während dies im Produktiv-Code zur korrekten Implementierung beitragen kann, signalisiert im Testfall das Auftreten einer derartigen Ausnahme meist einen Fehler, womit eine Behandlung selten sinnvoll ist. Um sich die Deklaration der Ausnahme in Hilfs- und Testmethoden zu sparen, werden diese Ausnahmen daher mit `FailureException` getunnelt, welche keine derartige Deklaration erfordert.

Hierzu wird die auftretende Checked Exception gefangen und eine `FailureException` ausgelöst. Der `FailureException` kann die ursprünglich aufgetretene Ausnahme übergeben werden, damit durch das Tunneln keine Informationen (wie der Stack Trace) verloren gehen.

11.3.2 Subklassen von `Node`

Die wichtigste Klasse im `classes`-Paket ist `IntNode`. Hierbei handelt es sich um eine von der generischen Klasse `BaseNode` abgeleitete Implementierung des Node-Interfaces mit der ganzzahlige Werte gespeichert werden können. `DifferentValuesIntNode`, `EvaluableTestNode` und `GeneratingIntNode` sind wiederum Subklassen von `IntNode`.

`IntNode` wird verwendet um die Implementierung der Minimax-Varianten auf ganzzahligen Bäumen zu testen. Das Testen mit ganzen Zahlen statt Schachstellungen vereinfacht das Schreiben und Debuggen von Tests drastisch. Da `BaseNode` eine generische Klasse ist und sowohl `Node` als auch `Evaluable` so weit wie möglich implementiert, teilen sich `IntNode` und die Produktivimplementierung `GameNode` sehr viel Code. Daher kann `GameNode` in vielen Tests durch `IntNode` ersetzt werden. Siehe hierzu auch die Beschreibung von `BaseNode` in Abschnitt 9.3.

`DifferentValuesIntNode` wird verwendet um sicherzustellen, dass (natürliche) Blattknoten gesondert ausgewertet werden. Ein natürliches Blatt ist hierbei ein Knoten, zu dem keine Kinder bestimmt werden können. Dies dient der Abgrenzung von Knoten, die zwar kinderlos sind, zu denen aber Kinder bestimmt werden könnten. Diese gesonderte Auswertung greift bei Matt und Patt, bei welchen keine Züge mehr möglich sind.

`EvaluableTestNode` dient dem Testen der `Evaluable`-Implementierung durch `BaseNode`. Instanzen von `EvaluableTestNode` zählen die Aufrufe gewisser Auswertungsmethoden, um das Caching ermittelter Bewertungen zu prüfen. Darüber hinaus ergeben sich andere Werte für Blätter als innere Knoten - dies ist bei `IntNode` nicht der Fall. Im Gegensatz zu `DifferentValuesIntNode` lässt sich die Bewertung von Blättern nicht gezielt steuern, sondern ergibt sich aus der Bewertung innerer Knoten.

`GeneratingIntNode` erweitert `IntNode` um die Funktionalität Kindknoten zu generieren. Hierdurch kann sichergestellt werden, dass Minimax-Varianten in der Lage sind Kindknoten selbständig zu generieren. Dies ist die Vorraussetzung dafür, dass der Spielbaum aufgebaut werden kann.

11.4 Paket `data`

Das Paket `data` ist das kleinste Test-Paket. Es besteht aus drei Bäumen zum Testen der Minimax-Varianten und einer Schachstellung inklusive aller Folgezüge im FEN-Format. Diese Daten werden im `tests`-Paket verwendet. Die Bäume sind eigene Klassen, können ohne Paramter instanziiert werden und erlauben eine Betrachtung des Werts jedes einzelnen Knotens.

11.5 Paket `helper`

Das Paket `helper` besteht aus mehreren Klassen, die Hilfsmethoden für Tests enthalten. Besonders hervorzuheben sind hierbei `MoveGeneratorHelper`, `Mirror`, `IntTreeEvaluationHelper`, `GameTreeEvaluationHelper` und `IntNodeHelper`.

11.5.1 `Mirror`

Ziel der `Mirror`-Klasse war es, existierende Zuggenerierungstests zu verstärken. Diese Tests bestehen aus einer Startstellung und den mögliche Folgestellungen. Die Startstellung wird der Zuggenerierung übergeben und die berechneten Folgestellungen werden mit

den erwarteten verglichen. Die `Mirror`-Klasse sollte es ermöglichen, jeden Testfall mit Weiß als aktiver Seite um einen äquivalenten Test mit Schwarz als aktiver Seite (und umgekehrt) zu ergänzen.

Hierzu macht die `Mirror`-Klasse es möglich Stellungen zu spiegeln - aus einer Stellung eine äquivalente Stellung mit vertauschten Farben zu erstellen. Dies wird erzielt, in dem das Brett horizontal gespiegelt und die Farbe jeder Figur geändert wird. Darüber hinaus müssen einige weitere Anpassungen vorgenommen werden. Beispielsweise werden die Rochade-Rechte von Schwarz und Weiß getauscht, sowie das En-Passant-Zielfeld und der zur Stellung gehörige Zug korrigiert.

Mit diesem Vorgehen können einzelne Stellungen gespiegelt werden. Versucht man nun jedoch Tests zu verstärken, indem man die Startstellung und jede Folgestellung spiegelt, tritt ein weniger offensichtlicher Fehler auf: Der Zugzähler der Folgestellungen muss korrigiert werden. Dieser wird nach jedem Zug von Schwarz um eins erhöht. Um einen Test mit Schwarz als aktiver Seite zu verstärken, wird die Startstellung gespiegelt und zu einer Stellung mit Weiß als aktiver Seite und dem gleichen Zugzähler. Die Folgestellungen der ursprünglichen Stellung haben einen erhöhten Zugzähler, da sie durch die schwarze Seite erzeugt wurden. Werden diese gespiegelt, wurden sie durch die weiße Seite erzeugt und dürfen deshalb den Zugzähler der Startstellung nicht erhöhen. Das Problem tritt analog mit Schwarz als aktiver Seite auf und wird gelöst, indem je nach aktiver Partei der Zugzähler in Folgestellungen um eins erhöht oder verringert wird.

11.5.2 `MoveGeneratorHelper`

Die Klasse `MoveGeneratorHelper` dient dazu, Tests für die Zuggenerierung möglichst kompakt schreiben zu können. Im Folgenden werden einige wichtige Methoden der Klasse beschrieben.

Die Methode `verifyPieceMoveGenerationWithFenStrings()` ermöglicht die Überprüfung der Zuggenerierung für eine spezifische Figur. Hierzu werden eine Stellung (Position), die Art der Figur (als byte), Linie und Reihe der Figur in der Stellung (als zwei ints) und eine Liste der erwarteten Folgestellungen im FEN-Format übergeben. Die Methode überprüft, ob das Bewegen der Figur genau die erwarteten Stellungen erzeugt. Zusätzlich wird die Überprüfung durch Benutzung der Klasse `Mirror` automatisch auch für die andere Partei durchgeführt.

Die Methode `verifyMoveGeneration()` ermöglicht eine kompakte Überprüfung der Zuggenerierung für eine gegebene Stellung. Hierzu müssen der Methode lediglich die Stellung im FEN-Format und eine Liste der erwarteten Züge übergeben werden. Einer derartige Liste kann mittels einer existierenden Engine wie beispielsweise Stockfish generiert werden. Auch hier wird die Zuggenerierung nicht nur für die übergebene Stellung, sondern auch für die gespiegelte Stellung getestet.

Folgender Code wird beispielsweise eingesetzt um (gemeinsam mit anderen Tests) sicherzustellen, dass keine ungültigen Rochaden generiert werden.

```
MoveGeneratorHelper.verifyMoveGeneration(
    "4k2r/8/8/6R1/8/8/8/4K3 b k - 0 1",
            "h8h1", "h8h2", "h8h3", "h8h4", "h8h5", "h8h6", "h8
               h7", "h8f8", "h8g8", "e8d7", "e8e7", "e8f7", "e8
               d8", "e8f8");
```

11.5.3 `IntNodeHelper`

Die Klasse `IntNodeHelper` bietet verschiedene Hilfsmethoden für ganzzahlige Bäume an. Unter anderem ermöglicht die Klasse das Erzeugen und Invertieren derartiger Bäume.

Die Methode `createIntNodeTree(int degree, int... values)` erzeugt einen Baum mit gefordertem Grad und spezifizierten Blattwerten. Die als `values` angebenen Werte bilden eine Blattebene. Für jeweils `degree` viele Knoten wird ein Elterknoten erzeugt. Dieser Vorgang wird wiederholt bis alle Knoten von einer gemeinsamen Wurzel aus erreichbar sind. In jeder Ebene ist höchstens der letzte Knoten nicht vollen Grades.

Ein Baum wird invertiert indem der Wert aller Blätter mit -1 multipliziert wird. Dies ist nützlich um die Auswertung eines Baumes automatisch für beide Parteien zu testen. Siehe hierzu den Abschnitt über `IntTreeEvaluationHelper`.

11.5.4 `IntTreeEvaluationHelper` und `GameTreeEvaluationHelper`

Die Klassen `IntTreeEvaluationHelper` und `GameTreeEvaluationHelper` ermöglichen das Testen mehrerer Minimax-Varianten mit geringem Mehraufwand. Im folgenden Abschnitt werden Methoden der Klassen beschrieben. Das Schreiben allgemeiner Tests für alle Varianten wird im Abschnitt 11.6 über das `tests`-Paket erläutert.

11.5.4.1 `IntTreeEvaluationHelper`

Die Klasse `IntTreeEvaluationHelper` unterstützt Tests auf ganzzahligen Bäumen. Dem Konstruktor von `IntTreeEvaluationHelper` wird ein `Supplier` übergeben, mit dem Instanzen von `TreeEvaluator<Integer>` erzeugt werden können. Diese sind Minimax-Varianten für ganzzahlige Bäume. Zusätzlich wird bei der Instanziierung angegeben, ob die Variante Iterative Deepening unterstützt. Eine Variante unterstützt Iterative Deepening, wenn sie bei der Auswertung des Baumes keine Knoten löscht. Diese Unterscheidung ist nötig, da ganzzahlige Bäume von Hand konstruiert sind und sich anders als Spielbäume nicht aus der Wurzel neu berechnen lassen.

Mittels `instantiateTreeEvaluator()` kann eine Minimax-Variante für einen Test instanziiert werden. Hierzu wird der bei der Instanziierung übergebene `Supplier` genutzt. Darüber hinaus bietet die Klasse Methoden an, um ganzzahlige Bäume auszuwerten und auf ein erwartetes Ergebnis zu prüfen. Unter Verwendung der Invertierungsfunktionalität von `IntNodeHelper` kann jeder Baum für beide Parteien getestet werden. Hierzu wird ein ganzzahliger Baum erzeugt und das Ergebnis bei Auswertung für eine Partei bestimmt. Multipliziert man nun die Blattwerte des Baumes mit -1 und ändert die aktive Partei, so ergibt sich das gleiche Ergebnis mal -1.

Der folgende Code prüft beispielsweise, ob die Auswertung eines `IntNodeAsymmetricTestTree` mit Schwarz als aktiver Seite für die Wurzel des Baumes den Wert -20 bestimmt. Weiterhin wird geprüft, ob die Auswertung des invertierten Baumes mit Weiß als aktiver Seite das Ergebnis 20 erzeugt.

```
verifyTreeAndInvertedTree(-20, 7, false, () -> new
   IntNodeAsymmetricTestTree());
```

Wird kein von Hand konstruierter Baum verwendet, kann die Erzeugung der Testdaten in einem Aufruf miterledigt werden. Hier wird ein binärer Baum der Tiefe 4 für die weiße Seite mit einem erwarteten Ergebnis von 3 ausgewertet. Der Baum hat die nach `true` angegebenen Blattwerte. Analog zu obigem Beispiel wird zusätzlich der invertierte Baum für die schwarze Seite mit dem Wert -3 getestet.

```
    verifyTreeAndInvertedTreeBinary(3, 4, true, -1, 3, 5, 1, -6,
       -4, 0, 9);
```

11.5.4.2 `GameTreeEvaluationHelper`

Die Klasse `GameTreeEvaluationHelper` unterstützt Tests auf Spielbäumen. Analog zu `IntTreeEvaluationHelper` wird ein `Supplier` übergeben, mit dem Instanzen von `GameTreeEvaluator` erzeugt werden können. Da Spielbäume vollständig aus der Wurzel berechenbar sind, ist keine Angabe über Kompatibilität zu Iterative Deepening benötigt.

`GameTreeEvaluationHelper` bietet Methoden an um Spielbäume auszuwerten und um zuzusichern, dass ein gespielter Zug Teil (oder nicht Teil) einer gegebenen Liste ist.

11.6 Paket `tests`

Das Paket `tests` enthält die unter Verwendung der anderen Pakete geschriebenen Tests.

11.6.1 `IntTreeEvaluationTest` und `GameTreeEvaluationTest`

Die Klassen `IntTreeEvaluationTest` und `GameTreeEvaluationTest` enthalten Tests für Minimax-Varianten auf Spielbäumen beziehungsweise ganzzahligen Bäumen. Die Klassen speichern jeweils eine Instanz von `IntTreeEvaluationHelper` beziehungsweise `GameTreeEvaluationHelper`. Diese werden bei der Instanziierung der Testklassen übergeben. Die Hilfsklasseninstanzen werden benutzt, um Tests zu schreiben, die auf beliebigen Varianten ausführbar sind. Hierzu werden `instantiateTreeEvaluator()` und andere Hilfsmethoden der beiden Hilfsklassen benutzt.

Um die in `IntTreeEvaluationTest` und `GameTreeEvaluationTest` enthaltenen Tests auf eine Variante anzuwenden, kann man diese einfach ableiten und einen entsprechend instanziierten `IntTreeEvaluationHelper` beziehungsweise `GameTreeEvaluationHelper` übergeben.

Um beispielsweise Alpha-Beta-Pruning zu testen, wurden folgende Klassen angelegt.

```
public class AlphaBetaGameTreeTest extends GameTreeEvaluationTest {

    public AlphaBetaGameTreeTest() {
        super(new GameTreeEvaluationHelper(() ->
            new GameNodeAlphaBetaPruning()));
    }

}
```

```
public class AlphaBetaIntTreeTest extends IntTreeEvaluationTest {

    public AlphaBetaIntTreeTest() {
        super(new IntTreeEvaluationHelper(() ->
            new GenericAlphaBetaPruning<Integer>(), true));
    }

}
```

`ExpensiveGameTreeEvaluationTest` ist eine von `GameTreeEvaluationTest` abgeleitete Klasse und erweitert diesen um Tests, deren Ausführung mit einigen

Minimax-Varianten zu lange dauert. Performante Varianten leiten ihre Tests von `ExpensiveGameTreeEvaluationTest` statt `GameTreeEvaluationTest` ab.

Sind für eine Variante spezielle Tests notwendig, so können diese einfach in der abgeleiteten Testklasse hinzugefügt werden. Dies wird unter anderem eingesetzt, um die Löschung von Knoten aus dem ausgewerteten Baum zu überprüfen. Im Fall von `StoringMoveOrderingSelfDestructingAlphaBetaPruning` wird das gezielte Beibehalten errechneter Knoten getestet.