Vergleichsschnittstelle für Bots in rundenbasierten 2-Spieler Spielen

Bachelorarbeit

Jan Schlenzka
400736

12. Juli 2024

Betreuer: Prof. Dr. Benjamin Blankertz
Dr.-Ing. Stefan Fricke

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

Kurzfassung

Im Fachbereich Neurotechnologie an der TU Berlin entstehen Bachloarbeiten zu Spielen und Algorithmen. Diese vergleichen oft mehrere Algorithmen, oder Varianten von Algorithmen. Trotz gleicher Spiele können diese nicht direkt miteinander verglichen werden. Das Ziel diese Arbeit ist es eine Schnittstelle zu schaffen um einen direkten Vergleich zu ermöglichen.
Aus diesem Ziel haben sich vier Anforderungen für die Schnittstelle abgeleitet: Metrikerhebeung, geringe Einschränkung für den Benutzer, geringer Aufwand für den Benutzer und Effizienz.

Dafür wurden bereits existierende Schnittstellen für Spiele betrachtet und anhand welcher Metriken in der Vergangenheit verglichen wurde. Basierend auf den vorhandenen Schnittstellen und unter Berücksichtigung der Anforderungen, wurde dann die Schnittstelle entwickelt. Zur Demonstration wurde ein Turnier mit der Schnittstelle durchgeführt und ein Bot aus einer vergangenen Arbeit mit der Schnittstelle integriert.
Im Kern wird das Prinzip von Remote Procedure Invocation genutzt um Kommunikation über die Standardinput-, und Outputstreams zu ermöglichen. Die Bots werden als nativ ausführbare Dateien aufgerufen und laufen im Rahmen eines Round-Robin Turniers teilweise parallel.
Die Schnittstelle wird es in der Zukunft erlauben, Bots aus verschiedenen Arbeiten miteinander zu verlgeichen.

Inhaltsverzeichnis

1 Einleitung **1**

2 Hintergrund **2**
2.1 Gomocup . . . 2
2.2 Ludii . . . 3
2.3 Metriken vergangener Bachlorarbeiten . . . 3
2.4 AlphaBeta-Suche . . . 4
2.4.1 Eingrenzen möglicher Züge . . . 6
2.4.2 Evaluationsfunktion . . . 6
2.5 Turnierdesign . . . 6
2.5.1 Round-Robin . . . 7
2.5.2 Knockout-Tunrier . . . 7
2.5.3 Schweizer-System . . . 7

3 Vergleichsschnittstelle **9**
3.1 Funktionale Anforderungen . . . 9
3.2 Designschwierigkeiten . . . 9
3.2.1 Diverse Programmiersprachen und Anforderungen . . . 10
3.2.2 Parallelität . . . 10
3.2.3 Kommunikation der Bots . . . 10
3.2.4 Metrikerhebung . . . 12
3.3 Turnierform . . . 13
3.4 Aufbau . . . 13
3.4.1 Ordnerstruktur . . . 13
3.4.2 CSVHandler . . . 13
3.4.3 Bot . . . 14
3.4.4 BotInstance . . . 17
3.4.5 Game . . . 17
3.4.6 Match . . . 17
3.4.7 Tournament . . . 19
3.5 Ablauf . . . 19
3.6 Kommunikation . . . 19
3.6.1 Turnier und Match . . . 19
3.6.2 Match und Bot . . . 22
3.6.3 Match und Game . . . 23

4 Beispielhafte Anwendung mit Gomoku **24**
4.1 AlphaBetaBot . . . 24

4.2 Turnier . . . 25
4.2.1 Erwartete Ergebnisse . . . 25
4.2.2 Ergebnisse . . . 25
4.3 Match mit Python Bot . . . 26

5 Diskussion **29**
5.1 Erfüllung der funktionalen Anforderungen . . . 29
5.2 Limitationen der Arbeit . . . 29

6 Fazit **31**

7 Lektionen **32**

Abbildungsverzeichnis

2.1 vollständiger Minimax Suchbaum mit Tiefe 4 und Verzweigungsfaktor von 2 5
2.2 Suchbaum aus Abbildung 2.1 mit AlphaBeta. Alpha links; Beta rechts 5
2.3 Round Robin Turnier mit 8 Teilnehmern. Färbung zur Veranschaulichung Drehung. . 7

3.1 Exemplarischer Ablauf eines Spiels mit 2 Spielern als hin und her 11
3.2 besipielhafter .csv Aufbau 12
3.3 exemplarische Ordnerstruktur 14
3.4 exemplarischer Ablauf eines statischen Turniers mit m Runden und $n*1$ Matches pro Runde 20

4.1 beispielhafte Konfiguration des Bots 24

Tabellenverzeichnis

2.1 Gefahrenwerte 6

3.1 zusätzliche Optionen und Argumente für das Turnier 19
3.2 Key-Value Paare für INFO Nachrichten 21

4.1 Konfiguration der Bots 24
4.2 Ergebnisse des Turniers zusammengefasst 25
4.3 Generierte Züge der Matches «3_2_1» und «3_2_2» 26
4.4 Erfasste Daten von Bot4 in Match <2_1_2> 27
4.5 Ergebnisse des Matchs zwischen AlphaBetaBot und ZERO 28

1 Einleitung

Wann immer unterschiedliche Ansätze für ein und dasselbe Problem gefunden werden, so stellt sich automatisch die Frage: Welcher ist besser? Bei algorithmischen Lösungsansätzen zur Zugfindung bei 2-Spieler Spielen ist diese Frage ebenso aufgetreten. So sind im Laufe der Zeit bei den Bachlorarbeiten an der TU-Berlin oft zwei Ansätze paarweise verglichen worden. Zwischen den Arbeiten jedoch lassen sich schwer eindeutige Beziehungen bilden.

Diese Arbeit befasst sich mit der Art und Weise, wie algorithmische Ansätze in der Vergangenheit verglichen worden sind. Dabei ist das Ziel eine Möglichkeit zu schaffen, die entstanden Bots, gegeneinander zu Vergleichen.

In Kapitel 2 wird ein Überblick über bestehende Möglichkeiten des Vergleichs zweier Bots gegeben. Zudem werden vergangene Arbeiten betrachtet. Dabei wird untersucht, welche MEtriken in der Vergangenheit genutzt wurden um Bots zu vergleichen. Auch wird die AlphaBeta-Suche vorgestellt. Diese wird später für einen beispielhaften Bot genutzt.

Kapitel 3 dreht sich ausführlich um die Vergleichsschnittstelle. Dabei werden zuerst funktionale Anforderungen Vorgestellt. Anschließend, werden die daraus entstehenden Herausforderungen vorgestellt und begründet gelöst. Der daraus hervorgehende Aufbau, Ablauf und Kommunikation wird abschließend präsentiert.

Zur Demonstration wird in Kapitel 4 ein beispielhaftes Turnier mit Gomoku und Bots mit AlphaBeta-Suche vorgestellt. Zusätzlich wurde ein in Python geschriebener Bot einer vergangenen Arbeit mit der Schnittstelle integriert.

Abschließend wird in Kapitel 5 die Schnittstelle diskutiert.

2 Hintergrund

Begriffe

Bot Alleinstehendes Programm, welches Spielzüge für ein bestimmtes Spiel generiert.

Spiel Rundenbasiertes 2-Spieler Spiel. (Gomoku, Schach, ...)

Match Durchführung eines Spiels mit 2 Bots als Spielern.

2.1 Gomocup

Das Gomocup Turnier, basiert auf dem namensgebenden Spiel Gomoku. Gomoku ist ein 2-Spieler Brettspiel auf einem $15x15$ Go-Brett. Im Verlauf des Spiels, platzieren die beiden Spieler abwechselnd, ihre Spielsteine auf dem Brett. Die Spielsteine werden hier auf den Kreuzungen der Brettlinien platziert, nicht auf den Feldern, wie zum Beispiel bei Tic-Tac-Toe. Ziel des Spiels ist es, genau fünf Steine in einer Reihe zu haben. Mehr als fünf werden nicht gewertet.[1]

Beim Gomocup treten im Gomokuspiel jedoch Bots, statt regulärer Spieler an. Die Bots benutzen hier einen Gomocup Manager, speziell für diesen Zweck ausgelget ist. Die Bots werden im Kontext auch oft als Gehirn (Brain) bezeichnet. Aktuell empfohlen wird der Pskvork Manager[2]. Kommunikation verläuft über Textdateien, oder den Standardin-/output. Bei der Variante mit Dateien, liest die AI bei jedem ausführen die aktuellen Positionen ein, berechnet den besten Zug und schreibt diesen das wieder als Datei zurück an den Manager. Dieses Protokoll, wird jedoch von dem Gomocupteam als obsolet angesehen.[3]
Das aktuelle Protokoll benutzt den Standardin-/output. Hier beinhaltet jeder Zeile In/-Output genau einen Befehl. Neben Definitionen für die einzelnen Befehle beinhaltet das Protokoll auch Definitionen für gewisse Rahmenbedingungen wie Dateinamenkonvetion, sowie Verhalten in gewissen Situationen. Die folgenden Befehle sind notwendig:

> START akzeptiert ein zusätzliches [size] Argument. nach erhalt dieses Befehl initialisiert das Gehirn ein leeres Spielbrett der Größe [size].
>
> BEGIN wird vom Manager an eins der Bots geschickt, um das Spiel auf einem leeren Brett zu beginnen. Der Bot antwortet mit seinem generierten Zug.
>
> TURN wird vom Manager mit den Argumenten [X],[Y] gesendet. In den Argumenten ist der Zug des Gegenspielers enthalten. Der Bot antwortet mit seinem generierten Zug.

[1] `http://gomokuworld.com/gomoku/1`, 19.02.2024
[2] `https://gomocup.org/download-gomocup-manager/`, 19.02.2024
[3] `https://gomocup.org/detail-information/`, 19.02.2024

INFO enthält zusätzliche Information vom Manager, welche vom Bot ignoriert werden kann. Die Information kommt in Forn von [key] [value].

BOARD ist alleinstehend als Befehl und kündigt ein neues Spielbrett an. Auf den Befehl folgend wird Information zum neuen Spielbrett in der Form [X],[Y],[field] gesendet. Jede solche Zeile gibt Information über den Zustand einen Spielstein an Position [X][Y]. Nach der Spielbrett Information folgt ein DONE vom Manager. Der Bot antowrtet daraufhin wie bei BEGIN, oder START mit seinem genrierten Zug.

END führt zum terminieren des Bots. Dabei sollten alle temporären Dateien gelöscht werden. Sollte der Bot zu lange brauchen um zu terminieren, so wird es vom Manager terminiert.

Neben diesen notwendigen Befehlen beinhaltet das Protokoll auch weitere optionale Befehle. Diese sind jedoch nicht wichtig um am Gomocup teilzunehmen.[9]
Im Gomocup existieren mehrere Gruppen basierend auf unterschiedlichen Regelsätzen. Standard, mit den Regeln wie, am Anfang des Kapitels beschrieben, sowie Freestyle. Bei Freestyle gewinnen mindestens fünf Steine in einer Reihe, statt genau fünf. Ab 2024 gibt es für Freestyle auch zwei Kategorien im Bezug auf Brettgröße. Eine mit $15x15$ und eine mit $20x20$.[4] Wenn im Verlauf der Arbeit von Gomoku die rede ist, wird hier explizit Freestyle auf einem $15x15$ Brett gemeint.

2.2 Ludii

Das Ludii General Game System ist Teil des `Digital Ludem Project` (DLP). Das Ziel des Projekts ist es nachzuvollziehen, wie Spiele sich historisch entwickelt haben und eine Art Lebensbaum von historischen Brettspielen zu entwickeln[5]. Dabei werden Spiele in atomare Grundbausteine zerlegt, mit denen sich das Spiel formal beschreiben lässt; sogenannte Ludems.
Ludems sind die zugrundeliegenden Ideen und Elemente eines Spiels.[11] Sind dabei typischerweise Deckungsgleich mit den zugrundeliegenden Regeln. So markiert bei TicTacToe der Spieler welcher am Zug ist, ein leeres Feld als eigenes. Ob dies durch einen farbigen Stein auf einem physischen Brett gemacht wird, oder durch ein Kreuz auf Papier ist für das Ludem irrelevant. Nach dem zerlegen eines Spiels in diese Grundlegenden Ideen, die DNA, ist es möglich Spiele auf Gemeinsamkeit zu vergleichen.[11]

Nachdem ein Spiel nun in Ludii übertragen wurde, kann es durch die vorgegebenen Java Klassen in ausführbaren Code compelliert werden. Ein weiteres der Hauptziele von Ludii ist es ein allgemeines Interface für Bots zu bilden. Dafür bietet Ludii, neben einigen Standard Bots, auf Auswertungsmöglichkeiten. [6]

2.3 Metriken vergangener Bachlorarbeiten

In der Vergangenheit wurden bereits einige Bots für 2-Spieler Spiel, im Rahmen von Abschlussarbeiten erstellt. Bei diesen Arbeiten wurden ebenfalls gewisse Metriken erfasst, wobei einige beliebter sind. Die folgenden Arbeiten wurden betrachtet:

[4] `https://gomocup.org/detail-information`, 19.02.2024

AlphaZero gegen MCTS: Eine Studie zur KI-basierten Spielstrategie in Gomoku[2]

Vergleich der Suchalgorithmen Alpha-Beta, MTD(f) und Best-First-Minimax für das Othello-Spie[3]

Systematischer Vergleich von Alpha-Beta und Monte-Carlo Tree Search für das Spiel Othello[10]

Varianten der Monte-Carlo Tree Search für das Brettspiel Gomoku unter Berücksichtigung der Sudden win/death Problematik[4]

Heuristische vs. stochastische Suche - Alphabeta Pruning und Monte Carlo Tree Search im Spiel der Amazonen[14]

Im folgenden werden leicht verallgemeinert die Metriken aufgezählt, welche in den Arbeiten erhoben wurden.

Zeit In jeder der betrachteten Arbeiten wurde die Zeit in mindestens einer Form gemessen, bzw. erfasst.

Gewinnrate In jeder der betrachteten Arbeiten wurde die Anzahl der gewonnenen Spiele, oder direkt eine Gewinnrate erfasst.

Speicher Die Speichernutzung wurde in einer Arbeit direkt erfasst und in einer weiteren in der Auswertung formal erwähnt.

Iterationen Anzahl der Iterationen, ist vom Algorithmus abhängig. In drei der Arbeiten wurde eine Metrik erfasst welche als Anzahl der Iterationen ausgedrückt werden kann. (zum Beispiel: evaluierte Züge, Suchbaumgröße)

KI-Kennzahlen In einer der Arbieten wurden mehrere KI-Kennzahlen, wie Fehlerrate erfasst. Diese sind sehr domainspezifisch.

Erfolgsrate In einer der Arbeiten wurde eine Erfolgsrate erfasst, bestimmte Spielsituationen zu erkennen.

Da sowohl Zeit, als auch Gewinnrate in jeder der vergangenen Arbeiten erfasst wurde, sollten diese im Rahmen der Schnittstelle in jedem Fall erfasst werden. Auch die Anzahl der Iterationen sollte standardmäßig erfasst werden, da sie in der Mehrheit der Arbeiten erhoben wirde.

2.4 AlphaBeta-Suche

Zur Demonstration der Vergleichsschnittstelle wird als Beispiel ein Gomokubot mit AlphaBeta implementiert. Dieser hat neben dem Standardalgorithmus auch einige domänenspezifische Anpassungen. Im allgemeinen ist AlphaBeta Suche eine Verfeinerung des Minimax Suchalgorithmus.[7]

Bei der Minimax-Suche wird rekursiver Tiefensuche der Baum der möglichen Zuge aufgebaut.[7] In Abbildung 2.1 ist ein solcher Suchbaum gezeigt. Hier ist der Verzweigungsfaktor 2. Das heißt bei jedem Zug bestehen 2 Möglichkeiten. Dadurch verdoppelt sich die Breite bei jedem Schritt. Auch

erreicht der Suchbaum nach 3 Zügen die Blätter, hat also insgesamt Tiefe 4. Die Größe des Suchbaumes ist also vom Verzweigungsgrad und der Anzahl der Züge abhängig. Daher limitiert man meist die Suchtiefe und gibt für die Blätter mithilfe einer Bewertungsfunktion eine heuristische Bewertung der Spielstellung wieder.

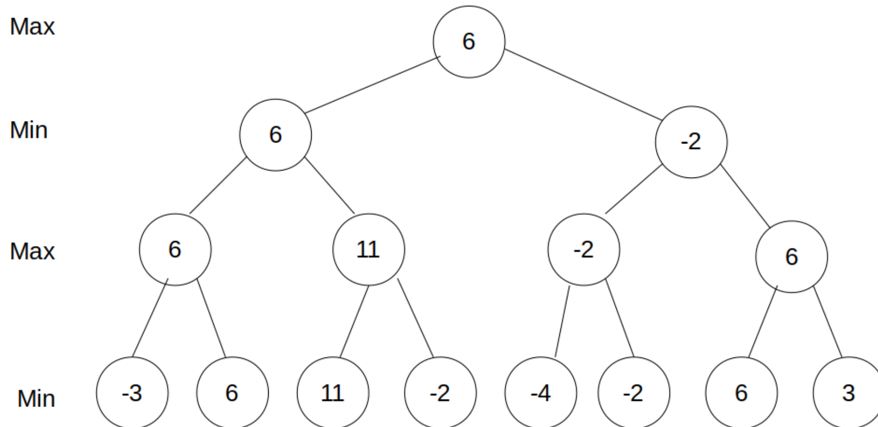

Abbildung 2.1: vollständiger Minimax Suchbaum mit Tiefe 4 und Verzweigungsfaktor von 2

Die AlphaBeta Suche versucht die Anzahl der durchsuchten Knoten zu verringern in dem der Suchbaum beschnitten wird. Dabei werden, für das Ergebnis irrelevante Teilbäume nicht besucht, unter der Annahme, dass optimal gespielt wird. In Abbildung 2.2 finden solche Beschneidungen, oder auch Cutoffs, in dem Suchbaum aus Abbildung 2.1 statt.[7]

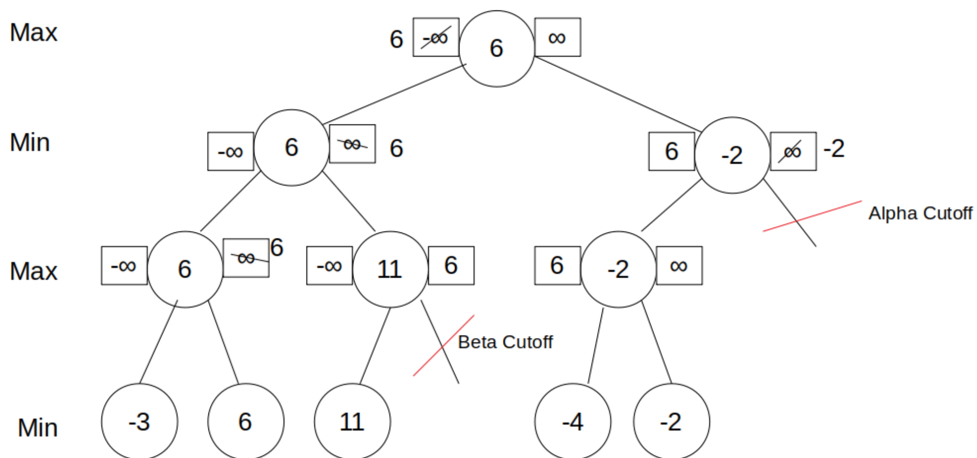

Abbildung 2.2: Suchbaum aus Abbildung 2.1 mit AlphaBeta. Alpha links; Beta rechts

Im linken Teilbaum findet ein Beta-Cutoff beim zweiten maximierenden Knoten in Tiefe 3 statt. Da der minimierende Elternknoten ein Ergebnis von 6 garantieren kann und der maximierende Knoten durch das erste Blatt bereits 11 erreichen kann, muss das zweite Blatt nicht mehr besucht werden, da

der Elternknoten niemals diesen Teilbaum wählen würde.
Im rechten Teilbaum findet hingegen ein Alpha-Cutoff am minimierenden Knoten in Tiefe 2 statt. Hier erhält der minimierende Knoten von seinem Kind den Wert -2 zurück, kann also diesen Wert garantieren. Da der maximierende Elternknoten jedoch das Ergebnis von 6 garantieren kann, wird er den linken Teilbaum nicht wählen, daher muss der restliche Teilbaum nicht mehr durchsucht werden.

2.4.1 Eingrenzen möglicher Züge

Bei Gomoku auf einem $15x15$ gibt es initial 255 mögliche Züge. Mit jedem Zug gibt es eine Zugmöglichkeit weniger. Ein vollständiger Suchbaum hat also einen Verzweigungsgrad von $255 - i$, wobei i die Anzahl der bereits gespielten Züge ist. Durchschnittlich dauert eine Runde 30 Züge. Daraus ergibt sich vollständige Suchbaumgröße von ca. 10^{70}.[1] Bei Gomoku haben Zuge, welche weit von bereits gesetzten entfernt sind, wenig Einfluss auf das Spielgeschehen. Gute Züge sind hingegen eher nah an bereits gesetzten Steinen.[13] Um also den Verzweigungsgrad zu verringern, werden nur Züge in direkter Nachbarschaft zu bereits gesetzten berücksichtigt.

2.4.2 Evaluationsfunktion

Bei einer begrenzten Suchtiefe hängt die Spielstärke stark von der gewählten Evaluationsfunktion ab.[7] Die hier gewählte Evaluationsfunktion, bewertet wie vorteilhaft ein bestimmter Zug ist. Dazu werden bestimmten Spielsituationen ein Gefahrenwert zugewiesen. Die Gefahrenwerte sind arbiträr gewählt. Jedoch wurde darauf geachtet, dass zwei mal 4 in einer Reihe den gleichen Gefahrewert produziert, wie 5 in einer Reihe. Wenn an zwei Stellen gleichzeitig ein Sieg gedroht erden kann, so ist das Spiel effektiv gewonnen. Wenn ein Blattknoten in der vorgegebenen Suchtiefe erreicht

Gefahr	Gefahrenwert
2 in einer Reihe	8
3 in einer Reihe	30
4 in einer Reihe	50
5 in einer Reihe	100

Tabelle 2.1: Gefahrenwerte

wird, so wird gibt die Evaluationsfunktion die Summe aller durch diesen Zug generierten Gefahren zurück.[13]

2.5 Turnierdesign

Das Design eines Turniers hat Einfluss auf die Anzahl an Spielen, die Unvorhersehbarkeit des Ergebnisses, sowie auf die generelle Fairness. Dabei gibt es drei generelle Formate, welche als Basis für alle weiteren angesehen werden. Diese sind Round-Robin (jeder-gegen-jeden), Knockout-Turnier, sowie Schweizer-System.[12]

2.5.1 Round-Robin

Bei einem Round-Robin Turnier spielt jeder gegen jeden. Dabei wird der Begriff genau dann verwendet wenn jeder die gleiche Anzahl an Spielen gegen jedes andere Team spielt. Dabei wird von einer geraden Anzahl an Teilnehmenden ausgegangen. Bei einer ungeraden Anzahl, erhält jede Runde ein anderes Team ein `Bye`, einen Aussetzen mit automatischen Sieg, sodass am Ende jedes Team einen automatischen Sieg erhalten hat. Die Anzahl an Spielen gegen einen bestimmten Gegner sein m. Dann ist bei $2n$ Teilnehmern die Gesamtzahl der Spiele $m(2n-1)$. Durch doppelte Matches, also ein gerades m, können Vorteile wie Erstzugvorteil ausgeglichen werden, indem jeder gleich oft den Vorteil erhält.
In Abbildung 2.3 zu sehen ist ein beispielhaftes Turnier mit 8 Teilnehmern. Die Matches sind immer zwischen den Spielern die übereinander stehen. Die Begegnungen können durch das Rotieren der Teilnehmenden, mit einem festen Team ermittelt werden. In der Theorie sind Round-Robin Tuniere

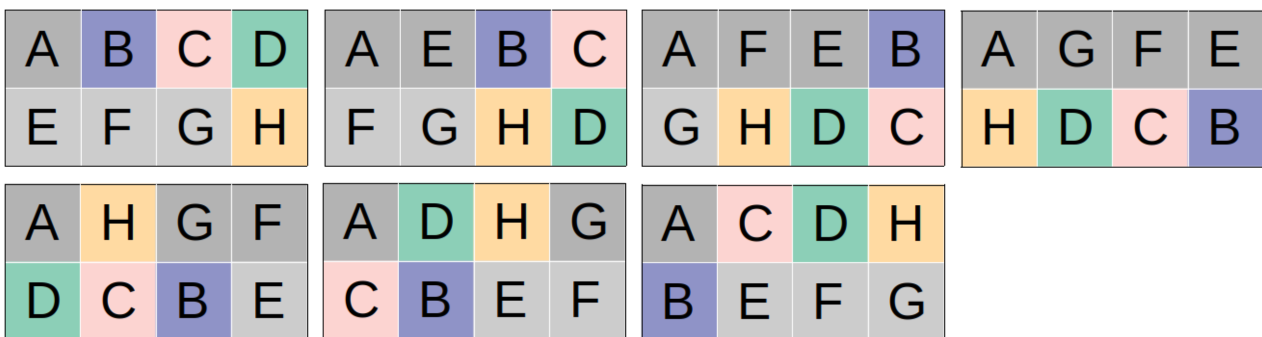

Abbildung 2.3: Round Robin Turnier mit 8 Teilnehmern. Färbung zur Veranschaulichung Drehung.

die fairste Turnierform, da kein Team aus dem Turnieraufbau einen Vorteil ziehen kann. Allerdings resultieren sie in sehr vielen Spielen. Bei professionellen Sportturnieren werden selten Round-Robin Turniere mit mehr als 20 Teilnehmern veranstaltet.[12]

2.5.2 Knockout-Tunrier

Ein Knockout Tunier ist ein Turnier, bei dem nach jeder Runde die Verlierer ausscheiden, bis nur noch der finale Gewinner übrig ist. Die Teilnehmerzahl ist hier 2^n für ein ganzzahliges $n \geq 1$. Da sich die Anzahl der Spieler jede Runde halbiert, erlaubt diese Form auch bei hoher Teilnehmerzahl einen schnellen Turnierverlauf. Da nach jeder Runde einer der beiden Kontrahenten ausscheidet, eignet sich diese Form nicht für Spiele, bei den ein Unentschieden möglich ist.[12]

2.5.3 Schweizer-System

Das schweizer System versucht ein Kompromiss aus Round-Robin und Knockout zu bilden. Dabei finden weniger Spiele statt. Teilnehmern ist es aber erlaubt im Verlauf den Turniers zu verlieren, oder ein Unentschieden zu erzielen. Dabei wird zu Beginn eine feste Anzahl an Runden festgelegt. Jede Runde werden Gegner ermittelt, die nach einer bestimmten Metrik (zum Beispiel Anzahl der Siege) eine ähnliche Stärke haben. Jedoch wird niemals mehrfach gegen den gleichen Gegner gespielt. Bei einem Turnier von $2n$ Teilnehmern und m Runden finden $n * m$ Spiele statt. Der finlae Gewinner, ist

der Teilnehmer, der am Ende den höchsten kumulativen Punktestand besitzt.[12]
Bei einer geraden Teilnehmerzahl, spielt jeder Teilnehmer in jeder Runde. Bei einer ungeraden Anzahl ist es auch hier möglich ein `Bye` mit aufzunehmen, um so wieder eine gerade Teilnehmerzahl zu erreichen. In diesem Fall leidet jedoch die Fairness, da nun eine Teil der Teams einen exklusiven Vorteil erhält.

3 Vergleichsschnittstelle

Die Vergleichsschnittstelle soll benutzt werden um den Vergleich verschiedener Bots für diverse 2-Spieler Spiele zu ermöglichen. Dabei handelt es sich um Bots welche im Rahmen von Bachlorarbeiten an der TU Berlin im Neurotechnologie Fachbereich entstehen.

3.1 Funktionale Anforderungen

Für das Interface ergeben sich mehrere funktionale Anforderungen, sowohl aus den Metriken welche benutzt werden sollen, als auch aus dem Anwendungsbereich. Im weiteren Verlauf der Arbeit werden diese Anforderungen mit ihrer entsprechenden Nummer referenziert.

(1)Metrikanwendung
Die Schnittstelle soll in der Lage sein möglichst viele der besprochenen Metriken zu erfassen. Dies ist nur zu einem gewissen Grad umsetzbar. Wie zuvor erläutert, kann Zeit von außen erfasst werden. Wenn jedoch Limits, wie eingeschränkte Zugzeit, benutzt werden sollen, muss dies vom Bot selbst umgesetzt sein. Allerdings soll das System in der Lage sein zu erkennen, wenn solche Einschränkungen verletzt worden sind.
(2)Geringe Einschränkung für den Nutzer
Um die Studierenden wenig einzuschränken, muss die Vergleichsschnittstelle möglichst agnostisch hinsichtlich der verwendeten Programmiersprache und Implementationsart sein. Auch soll den Studierenden eine möglichst freie Wahl hinsichtlich der Spiele ermöglicht werden.
(3)Geringer Aufwand für den Nutzer
Die Benutzung soll mit möglichst geringem Arbeitsaufwand verbunden sein, da die Bachlorarbeit der Studierenden im Vordergrund steht. Dies umfasst sowohl den reinen Zeitaufwand, als auch Komplexität des Interfaces für den Nutzer.
(4)Effizienz
Die Schnittstelle soll möglichst effizient sein um auch größere Vergleiche mit mehreren Bots realistisch zu ermöglichen.

3.2 Designschwierigkeiten

Die größte Herausforderung im Design des Interfaces bildet die geringe Einschränkung für den Nutzer. Dies ist auch der Grund, dass weder Ludii, noch Gomocup, sich als Interface eignen. Während Gomocup die Studierenden auf ein komkretes Spiel einschränkt, Gomoko, ist Ludii in dieser Hinsicht allgemeiner. Allerdings sind Bots für Ludii auf Java beschränkt, was ebenfalls in Konflikt mit dem Designziel (2) steht.

3.2.1 Diverse Programmiersprachen und Anforderungen

Da jede Einschränkung für den Nutzer vermieden werden soll, gerade was die Wahl von Sprache und Tools für ihre Arbeit betrifft, muss mit potentiell diversen Anforderungen für die verschiedenen Bots gerechnet werden. Um dies zu lösen könnte man die Bots grundsätzlich in, vorkonfigurierten Dockern, oder mit Werkzeugen wie Apache Maven, aufsetzen. Einen solchen Anspruch zu erheben, würde jedoch mit (3) in Konflikt stehen, solange es eine Lösung mit weniger Aufwand gibt.
Diese Lösung ist das Nutzen von nativ ausführbaren Dateien. Dieser Lösungsweg ist inspiriert vom Gomocup. Sprachen mit einem Compiler erstellen automatisch ausführbare Dateien (native-executable), welche dann unabhängig ausgeführt werden können. Sprachen wie Java, oder Python, welche keinen regulären Bytecode erstellen, oder einen Interpreter benutzen, können mit Tools als ausführbare Dateien zur Verfügung gestellt werden.[1][2] Diese ausführbaren Dateien können dann, wenn vom Interface benötigt, aufgerufen werden. Daraus leitet sich die erste Anforderung für die Bots ab: Der Bot muss als eigenständige ausführbare Datei zur Verfügung gestellt werden.

3.2.2 Parallelität

Den Anspruch für Parallelität leitet sich aus der (4)Effizienz ab. Dafür muss das gesamte Turnier, idealerweise, in viele kleine Jobs geteilt werden, welche unabhängig voneinander berechnet werden können. Die Ergebnisse dieser Jobs müssen dann wieder akkumuliert werden.

Um das ohne Einschränkungen zu ermöglichen, müssten die Bots threadsicher sein. Dies steht jedoch mit dem (3), dem möglichst geringen Aufwand, in Konflikt. Die Arbeiten im Fachbereich beschäftigen sich mit Algorithmen. Es kann nicht von jedem Studierenden verlangt werden, seinen Bots threadsicher zu implementieren.

Dieses Problem kann bis zu einem gewissen Grad bei einem Round-Robin Aufbau gelöst werden. Da jeder gegen jeden verglichen wird, kann der Vergleich in $n - 1$ Runden aufgeteilt werden. In jeder Runde gibt es $n/2$ Spiele mit jeweils 2 einzigartigen Partnern. Diese Spiele können parallelisiert werden, ohne das Parallelität und somit Threadsicherheit von den Bots gefordert werden muss.

3.2.3 Kommunikation der Bots

Eine zentrale Herausforderung ist die Kommunikation der Bots mit Interface und dadurch untereinander. Bei jeder der Ansätze, bleibt für die Studierenden die Anforderung, Strings einzulesen und zu bearbeiten. Für die Kommunikation gibt es einige grundsätzlich konzeptionelle Möglichkeiten:

1. **Messaging** Eine Anwendung schickt eine Nachricht in einen gemeinsamen Kanal. Eine andere Anwendung kann diese dann, zu einem anderen Zeitpunkt lesen. Die Anwendungen müssen im vornherein sich auf den Kanal und das Format der Nachricht einigen. Diese Kommunikation ist asynchron.

2. **Shared Database** Mehrere Anwendungen nutzen eine gemeinsame Datenbank, verortet an einem einzigen physischen Ort. Da es nur einen Speicherort gibt, ist kein Datentransfer nötig.

[1] `https://pyinstaller.org/en/stable/`, 5.Juli.2024
[2] `https://www.graalvm.org/latest/reference-manual/native-image/`, 5.Juli 2024

3. **Remote Procedure Invocation** Eine Anwendung öffnet ein Teil seiner Funktionalität nach außen, sodass diese von einer anderen aufgerufen werden kann. Diese Kommunikation ist synchron.

4. **File Transfer** Dateien werden von einer Anwendung abgespeichert um von einer anderen genutzt zu werden. Dabei müssen sich die Anwendungen im vornherein auf ein Protokoll, sowie Speicherort einigen.

[8] Die Kommunikation findet an zwei Stellen statt. Zwischen Turnier und den einzelnen Matches, sowie zwischen Match und den Bots der Studierenden. Die Matches finden wie bereits in 3.3.2 dargelegt teilweise parallel statt. Sowohl **Messaging**, als auch **Shared Database** benötigen, zusätzliche Komponenten welche potentiell mit (3) in Konflikt stehen. Übrig bleibt **File Transfer** und **Remote Procedure Invocation**. Die $n/2$ parallelen Runden sind unabhängig voneinander und können daher von asynchroner Kommunikation profitieren. Da **File Transfer** grundsätzlich asynchron ist, wird es vorgezogen. Bei der Kommunikation zwischen Match und Bot, lässt sich kein nennenswerter Nutzen aus asynchroner Kommunikation ziehen, wie in Abbildung 3.1 zu sehen. Da die Spiele rundenbasiert abwechselnd stattfinden, stehen die Bots im ständigen Austausch mit dem anderen Spieler und blockieren effektiv, bis der Mitspieler seinen Zug gemacht hat. Um (3) möglichst nachzukommen wird dennoch erst einmal auch die Kommunikation zwischen den Bots mit **File Transfer** gelöst, statt demr standardmäßig synchronen **Remote Procedure Invocation**. Dadurch kann die gesamte Software ein einheitliches Kommunikationsschema haben.

Im ersten Entwurf orientiert sich die Kommunikation stark am veralteten Protokoll des Gomocup.

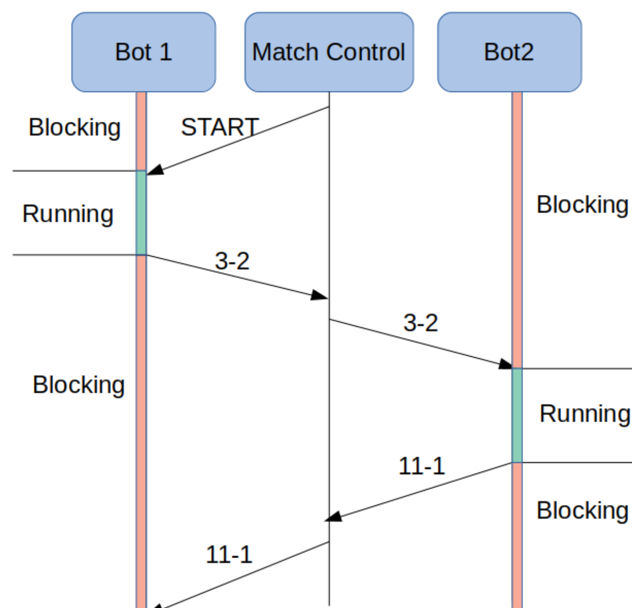

Abbildung 3.1: Exemplarischer Ablauf eines Spiels mit 2 Spielern als hin und her

Hier werden, die Bots abwechselnd ausgeführt, und legen ihr generierten Zug in einer lokalen Datei ab. Die Kommunikation erfolgt hier rein über lokale Dateien. Schreib/Lese Konflikte werden hier durch das abwechselnde Ausführen verhindert. Beim Aufruf, kann der Bot zu Beginn einen Zustand aus lokalen Dateien laden, falls nötig. Dies stellt auch das Hauptproblem für diese Lösung dar. Wenn

ein hypothetischer Algorithmus eine große Struktur nutzt und diese zwischen Zügen persistent hält, dann ist der Overhead zum Initialisieren bei jedem Start des Bots entsprechend groß. Eine mögliche Lösung wäre binäre Serialisierung von Daten zum speichern. Dies ist jedoch nicht in allen Programmiersprachen, ohne größeren Aufwand möglich. Somit Steht diese Lösung im Konflikt mit (3) und/oder (4).
Der zweite Entwurf, löst dieses Problem, indem er sich die synchrone Natur der Bots zu Nutze macht. Die Bots werden einmal gestartet und laufen bis Abschluss des Spiels weiter. Dies erlaubt es Datenstrukturen im Heap zu lassen und entfernt den Overhead des Initialisierens. Die Kommunikation erfolgt mit Prinzipien von **Romote Procedure Invocation**. Beide Bots werden von einem Match aus gestartet. Dem Match stehen die standard Input-und Outputstreams als Ressourcen zur Verfügung. Dies erlaubt es dem Match einen Input an den Bot zu senden, welcher wie eine Kommandozeileneingabe behandelt wird.
Der Match wiederum kommuniziert ebenfalls mittels **Remote Procedure Invocation** mit dem Turnier. Das Problem der synchronen Natur, kann über eine parallele Implementation gelöst werden, in der jedes Match auf einem eigenen Thread läuft. Die Kommunikation erfolgt konkret über Futures[3]. Dies steht nicht im Konflikt mit (3) wie in 3.2.2 dargelegt und ermöglicht es die Kommunikation einheitlich zu halten.
In jedem Fall ist ein Protokoll von Nöten, welches die Kommunikation der verschiedenen Komponenten ermöglicht.

3.2.4 Metrikerhebung

Eine offene Frage bleibt, wer die Verantwortung hat, die Metriken zu messen. Im Idealfall, misst die Schnittstelle alle Metriken, ohne Aufwand für den Nutzer. Dies ist jedoch nicht Umsetzbar. Gewisse Metriken wie Zeitaufwand oder Gewinnrate können von außen gemessen werden. Andere Metriken wie Iterationstiefe, oder besuchte Knoten im Suchbaum können nicht gemessen werden, wenn der Bot wie eine Blackbox behandelt wird. Daher folgt der Grundsatz: Alles was nicht von außen gemessen werden kann, muss vom Bot selbst erfasst werden. Dadurch wird der Aufwand erhöht, was im Konflikt mit (3) steht, ist jedoch ein notwendiger Kompromiss für die Umsetzung von (1).
Wenn jeder Bot selbst für einen Teil der Metrikerfassung verantwortlich ist, so muss die Art und Weise vereinheitlicht werden, damit die Schnittstelle flexibel bleibt. Hierzu sollten alle Metriken in ein leicht einzulesendes Format geschrieben werden. Dazu werden .csv Dateien genutzt.

```
Turn;Time;Iterations;...
6-3;567;1350;;...
```

Abbildung 3.2: besipielhafter .csv Aufbau

Um eine Verarbeitung zu erleichtern und Vergleiche mit in der Vergangenheit gesammelten Daten zu ermöglichen, sollte die Struktur der .csv Datei möglichst gleich bleiben. Wenn Metriken hinzukommen, sollte folgendem Ablauf gefolgt werden: Lässt sich die neue Metrik durch eine bereits

[3] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html, 20.05.2024

vorhandene ausdrücken? (Anzahle besuchter Knoten -> Iterationen) Falls, nein wird die neue Metrik zum Kopf hinzugefügt.

3.3 Turnierform

Im Kontext eines digitalen Turniers zwischen Bots, ist die Unvorhersehbarkeit des Ergebnisses zu vernachlässigen, da es nicht darum geht, Spannung oder Zuschauer-/Sponsoreninteresse zu erhalten. Die Anzahl der Spiele hat einen direkten Einfluss auf (4) und ist eine direkte Repräsentation der Laufzeit des gesamten Turniers. Der Einfluss auf die Fairness des Turniers ist der wichtigste Aspekt, da sich dieser direkt auf (1) auswirkt; spezifisch auf die Gewinnrate.
Daher implementiert die Schnittstelle ein Round-Robin Turnier mit variabler Anzahl der Matches pro Begegnung.

3.4 Aufbau

Die Schnittstelle ist in einige Klassen unterteilt, welche im folgenden erklärt werden. Dabei ist 3.4.3 die Vorgabe eines Bots in Java. Diese wird beispielhaft zum Erklären genutzt, ist jedoch selbst nicht Teil der Schnittstelle. Die einzelnen Bots können in einer belibigen Programmiersprach verfasst sein, solange sie das Protokoll in 3.6.2 implementieren. Die hier aufgeführten Klassen, Funktionen und Attribute haben keinen Anspruch auf Vollständigkeit und repräsentieren nennenswerte Teile der vollständigen Schnittstelle. Die im Code gegebenen Kommentare sind javadoc kompatiebel und können genutzt werden um eine Dokumentation zu erzeugen.

3.4.1 Ordnerstruktur

Damit die Schnittstelle funktioniert, muss die vorgegebene Ordnerstruktur eingehalten werden, besipielhaft in Abbildung 3.3. Dabei ist jeder Bot in einem Ordner mit dem gleichen Namen wie die ausführbare Datei. Dieser Bot-Ordner ist das Arbeitsverzeichnis für den Bot. Hier kann neben dem Bot selbst, Konfigurationsdateien, Logs, oder benötigte Daten gespeichert werden. Auch wird hier ein mögliches Startbrett als Datei abgelegt. Im darüberliegenden Turnier Ordner befindet sich die ausführbare Datei des Turniers und von Match. Generierte Log-Dateien werden ebenfalls hier direkt abgelegt.

3.4.2 CSVHandler

Dies ist eine vorimplementierte Hilfsklasse, die bei der Metrikerhebung helfen soll. Sie wird von Match und Turnier zum schreiben, der von ihnen generierten .csv Dateien genutzt. Diese Klasse enthält keine von außen zugänglichen Attribute.

public CSVHandler(String[] header, String delimiter)

Im Konstruktor wird ein String-Array übergeben, welches die Spaltennamen für die generierte Tabelle enthält. Auch wird hier das benutzte Trennsymbol übergeben.

```
~\Turnier
|--Tournament.jar
|--Match.jar
|--Bot1
|   |--botfile1.cfg
|   |--Bot1.exe
|   |--match1.log
|--Bot2
|   |--botfile1.cfg
|   |--Bot2.exe
|   |--match1.log
|--default.log
|--points.csv
```

Abbildung 3.3: exemplarische Ordnerstruktur

public void initializeFile(String filename, String filePath) throws IOException

Mit dieser Funktion wird eine neue .csv Datei angelegt und die Spaltennamen werden hineingeschrieben. Wenn mehrere Dateien mit gleicher Kopfzeile geschrieben werden soll, muss so nicht ein neues Objekt angelegt werden.

public void addValue(String column, String value)

Speichert das übergebene Werte-, Spalten-Paar für die nächste Zeile ab. Wenn bereits ein Wert für dieser Spalte für die aktuelle Zeile gespeichert ist, so wird dieser überschrieben.

public void buildRow()

Fasst alle zwischengespeicherte Werte für die aktuelle Zeile zusammen und speichert die Zeile ab. Daraufhin wir der Speicher der aktuellen Zeile geleert.

public void writeRows() throws IOException

Schreibt alle gespeicherten fertigen Zeilen in die .csv Datei und leert den Speicher fertiger Zeilen.

public void close() throws IOException

Gibt die .csv Datei als Resource wieder frei.

3.4.3 Bot

Die Klasse Bot ist in diesem Fall als abstrakte Klasse implementiert, bei der die nötige Funktionalität für Kommunikation bereits implementiert ist.

Attribute

Die Attribute lassen sich grob in zwei Kategorien aufteilen. Zum einen die allgemein zur Funktionalität nötg/hilfreichen. und die zur Konfiguration. Relevant für die Funktionalität sind hier gerade die Booleans $started$ und $shutdown$, welche als Flags für den internen Zustand genutzt werden. Nach dem der Bot begonnen hat Züge zu generieren, sollte $sterted = true$ gesetzt werden. Ab diesem Zeitpunkt darf der Bot nur noch Spielzüge generieren und seine Konfiguration nicht mehr ändern. Wenn $shutdown == true$ ist, soll der Bot terminieren. Auch ist $loop$ relevant. Dieses Attribut wird jedes mal inkrementiert, wenn ein Fehler bei der Kommunikation mit dem Match auftritt. Wenn mehr als 10 Fehler aufgetreten sind, wird davon ausgegangen, dass das Match terminiert hat und der Bot verwaist ist. In diesem Fall soll der Bot terminieren.
Die Attribute zur Konfiguration, besitzen alle Standardwerte. Diese können durch Nachrichten vom Match überschrieben werden.
Die vorgegebenen Attribute zur Konfiguration, sowie die boolischen Attribute für den internen Zustand besitzen Getter-Methoden, jedoch keine Setter-Methoden.

public abstract void generateMove(String move) throws InputMismatchException

In Implementation des Bots, durch die Studierenden, ist hier der Großteil der Logik. Sie wird aufgerufen, wenn der Bot eine $TURN$ Nachricht erhält. Die Funktion akzeptiert als Parameter den String $move$. In diesem ist der vorangegangene Zug des Gegenspielers gespeichert. Benutzt die vorgegebene $sendMessage()$ um den generierten Zug zu senden.

public abstract void openingMove()

Wird aufgerufen, wenn der Befehl $START$ erhalten wird. Sendet den Eröffnungszug. Um vordefinierte Eröffnungszüge leichter einzubauen, ist diese Funktion separiert von $generateMove()$.

public abstract void parseBoard(String board_name)

Diese Funktion dient der Option das Spiel in beliebigen Spielsituationen zu starten. Sie sollte aufgerufen werden, wenn der Bot die entsprechende Information vom Match erhält. Als Parameter wird der String $board_name$ übergeben. Dies ist der Dateiname mit Dateiformat, für den Startzustand. Dateiformat und Aufbau der Datei sind vom konkreten Spiel abhängig. Sobald jedoch das Erste mal ein Format für ein Spiel festgelegt wurde, sollte dieses für Kompatibilität eingehalten werden.

public abstract void run()

Hier sollte die Haupt-Schleife des Bots enthalten sein. Im Fall, dass das $shutdown == True$ ist, muss der Bot diese Schleife verlassen und terminieren. Hier beispielhaft eine simple Form:

```
@Override
public void run() {
    while (true) {
        waitOnMessage();
        if (isShutdown())
```

```
                break;
            }
            logger.log(Level.INFO, "Shutting down normally");
        }
```

public void waitOnMessage()

Eine der vorimplimentierten Funktionen. Hier wird auf neuen Input blockierend gewartet. Sobald eine Nachricht eingegangen ist, wird diese verarbeitet und die dafür nötigen Aufrufe getätigt. Diese Verarbeitung passiert über einen switch-case. Dafür wird die Nachricht an Leerzeichen in ein Array *line* aufgeteilt. Hier beispielhaft ein paar der Fälle:

```
switch (line[0]){
    case "START":
        openingMove();
        started = true;
        break;
    case "TURN":
        generateMove(line[1]);
        started = true;
        break;
    case "END":
        shutdown = true;
        logger.log(Level.FINE, "set shutdown flag");
        break;
    case "INFO":
        if (started){throw new IllegalStateException("Game in progress");}
        switch (line[1]){
            case "timeout_turn":
                timeout_turn = Integer.valueOf(line[2]);
                logger.log(Level.CONFIG, "set timeout_turn: " + timeout_turn);
                sendMessage("OK");
                break;
```

Nach dem Aufruf von $START$, oder $TURN$ wird das entprechende Attribut auf $true$ gesetzt damit der interne Zustand konsistent mit dem vom Match angenommenen bleibt. Im Anschluss wird die abstrakte Funktion $generateMove()$ mit dem entsprechenden Input aufgerufen. Ähnlich wird bei END ebenfalls der interne Zustand angepasst.
Im Fall von $INFO$ wird erst der interne Zustand überprüft, ob aktuell an der Kofiguration Änderungen vorgenommen werden können. Falls ja, wird für das entsprechende Attribut die Änderung vorgenommen und dann, dem Protokoll folgende ein OK gesendet.

public void sendMessage(String message)

Ebenfalls vorgegeben, schreibt diese Funktion den mitgegebenen String auf std.out. Zudem wird für das entsprechende Ereignis ein Eintrag im Log gemacht.

public String readBoardFromFile(String board_name) throws IOException

Vorgegebene Hilfsfunktion, die die Datei mit dem übergebenen Namen einliest und den Inhalt aus String zurückgibt.

3.4.4 BotInstance

Diese Klasse ist ein Wrapper für die ausgeführten Bots. Der Input und Output des Bots wird durch diesen Wrapper verwaltet und weitergeleitet.

public BotInstance(ProcessBuilder builder, String name)

Der Konstruktor erhält das *ProcessBuilder* Objekt, sowie den Namen des Bots. Dies wird in entsprechenden Attributen hinterlegt.

public void start() throws IOException

Hier wird der Prozess gestartet und ein Reader und Writer für die Kommunikation mit dem Bot erstellt. Auf diese können mit Getter-Methoden zugegriffen werden.

3.4.5 Game

Ein Interface für die Spielinstanz. Dieses Interface besitzt nur 2 Methoden, welche vom Spiel implementiert werden müssen. Von Match wird über diese beiden Funktionen mit der Objekt kommuniziert.

Damit die Schnittstelle funktioniert benötigt sie die Spielimplementation als Bibliothek, genannt Game.jar. Die Game.jar muss die beiden folgenden Dateien enthalten: Game.java und GameImplementation.java. Game.java ist das hier aufgeführte Interface über das die Schnittstelle kommuniziert und GameImplementation.java ist die eigentliche Implementation des Spiels. GameImplementation.java muss für jedes Spiel neu geschrieben werden und gegebenenfalls, neue .jar Dateien für Match und Tournament generiert werden.

public String evaluateMove(String move) throws IllegalArgumentException

Nimmt den zu prüfenden Spielzug als String an. Daraufhin sollte der Zug ausgeführt werden und Information über die Änderung des Spielzustandes als String zurückgegeben. Ist der Zug illegal, so soll eine Exception geworfen werden.

public void setupBoard(String board)

Setzt den internen Zustand des Spiels so, dass es dem als String übergebenen Brett entspricht.

3.4.6 Match

Die Klasse Match beinhaltet das Programm für das Spielen eines einzelnen Matches zweier Bots. Um Parallelität mehrerer Matches zu ermöglichen, implementiert die Klasse das *Callable* Interface. Die

Klasse hat jedoch auch eine *main* und kann über die Kommandozeile genutzt werden. Die Struktur von Match ähnelt der der Bot-Vorgabe. So besitzt sie eine sehr analoge $waitOnMessage()$.

Game

Damit die Klasse funktioniert benötigt sie die Spielimplementation als Bibliothek. Die Game.jar muss Die beiden folgenden Dateien enthalten: Game.java und GameImplementation.java. Game.java ist ein Interface über das die Schnittstelle kommuniziert und GameImplementation.java ist die eigentliche Implementation des Spiels.

Attribute

Bemerkenswert sind hier die Attribute $HashMap < String, BotInstance > bots$ und $Gamegame$. In $bots$ werden Referenzen auf die Bot-Wrapper zusammen mit dem Namen gespeichert. In $game$ wird die Spielinstanz vom Match gespeichert. Die Instanz ist von der Klasse $GameImplementation$ welche das Interface $Game$ implementiert.

private void intitalizeBots()

In dieser Funktion werden die Bots als Prozesse gestartet und die $INFO$ Nachrichten werden verschickt. Die vom Turnier erhaltenen Informationen werden hier an die Bots weitergereicht. Zusätzlich wird die Instanz von Game erstellt und initialisiert.

```
game = new GameImplementation();
if (initial_board.compareTo("")!=0) setInitial_board();
...
for (BotInstance current:bots.values()) {
        //send MatchName
        commandBot(current, "INFO match_name "+match_name);
        //send further INFO
        ...
    }
```

public String commandBot(BotInstance bot, String message)

Alle Befehle die an die Bots gesendet werden, nutzen diese Fubktion. Hier wird die Nachricht verschickt und von außen auf die Antwortzeit geachtet. Die vom Bot erhaltene Antwort wird als String zurückgegeben. Wird die Zeitbegrenzung mit Toleranz überschritten, so wird der Fehler im Log erfasst und als Antwortnachricht ein leerer String übergeben.

public HashMap<String,Integer> runMatch(String startingBot)

Dies ist die Hauptfunktion, in der der Großteil der Programmlogik sitzt. Die generierten Züge werden wie in Abbildung3.4 ausgetauscht. Zusätzlich wird jeder Zug in der SPielinstanz ausgeführt und auf Legalität und Sieg, oder Niederlage geprüft. Ist ein Zug illegal, so wird das Spiel mit einem Sieg für den Gegenspieler beendet. Das gleiche geschieht, wenn bei dem Aufruf von $commandBot()$ eine leere Antwort erhalten wird.

3.4.7 Tournament

Dies ist die Hauptklasse des Turniers. Die Attribute werden zum speichern von übergebenen Arbumenten genutzt. Diese Klasse benutzt das «Apache Commons CLI 1.6.0» zur Verarbeitung der Kommandozeileneingabe.[4]

public static void main(String[] args)

Das Turnier nimmt als Argumente alle am Turnier beteiligten Bots. Dazu werden die Namen der Bots mit Leerzeichen getrennt übergeben. Zusätzlich können weiter Argumente mit den in Tabelle3.1 gegebenen Optionen übergeben werden.
Anschließend werden die einzelnen Paarungen für die Spiele als Round-Robin Turnier ermittelt und dann gemäß dem Ablauf durchgeführt.

Option	übergener Wert	Beschreibung
-B	initialBoard	Name der Datei für ein alternatives Spielbrett
-g	gracePeriod	Toleranz bei Zeitüberschreitung der Bots in ms
-n	name	Name des Turniers
-t	timeout	Zeitbeschränkung der Bots in ms
-m	matches	Anzahl der Spiele pro Begegnung in einer Runde
-h		druckt Hilfsnachricht

Tabelle 3.1: zusätzliche Optionen und Argumente für das Turnier

3.5 Ablauf

Ein Turnier ist in mehrere Runden unterteilt. In jeder Runde finden n Matches, k mal mit wechselndem Startspieler, gegebenenfalls parallel, statt. In Abbildung 3.4 ist ein solches Turnier zu sehen. Zu Beginn werden alle Begegnungen ermittelt. Im Anschluss beginnen die einzelnen Matches der Ersten Runde. Innerhalb eines Matches, werden die von den Bots generierten Züge überprüft und dann gegebenenfalls weitergeleitet.
Bei einer anderen Tunrierform, wie Schweizer-System, müssten dieTurnier Begegnungen nach jeder Runde neu ermittelt werden.

3.6 Kommunikation

Die Kommuniaktion der einzelnen Komponenten unterliegt bestimmten Regeln, welche im folgenden gezeigt werden.

3.6.1 Turnier und Match

Das Turnier kann auf 2 Wegen mit den Matches kommunizieren. Einmal über einen direkten Aufruf, oder über std.in und std.out als Schnittstellen. Standardmäßig kommuniziert das Turnier über einen

[4] https://mvnrepository.com/artifact/commons-cli/commons-cli/1.6.0, 05.Juli 2024

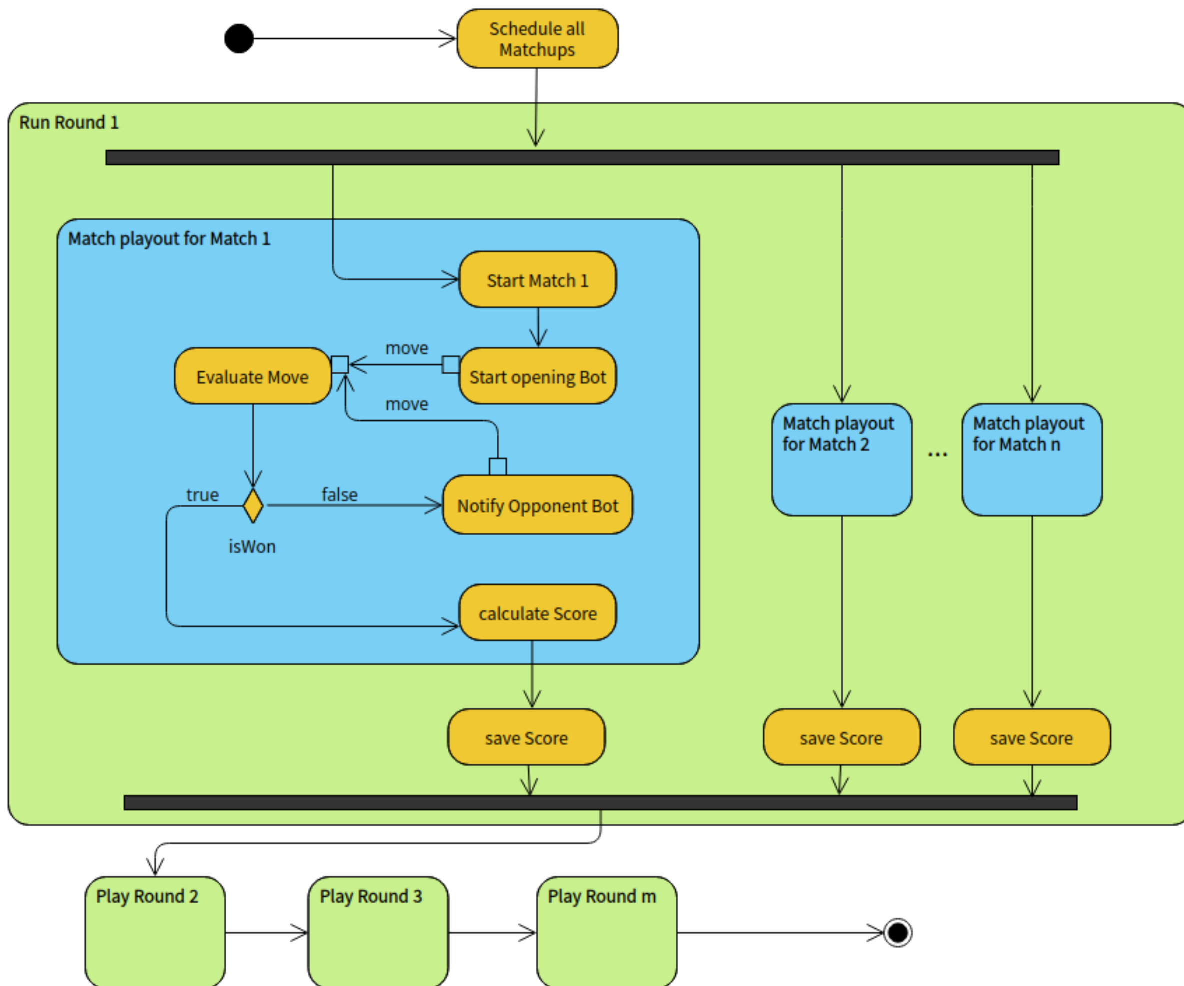

Abbildung 3.4: exemplarischer Ablauf eines statischen Turniers mit m Runden und $n*1$ Matches pro Runde

direkten Aufruf. Jedes Match repräsentiert dabei eine Instanz des Spiels. Zum simulieren einzelner gezielter Spiele eignet sich der Aufruf über die Kommandozeile unter Benutzung des Protokolls.

Direkter Aufruf - Kommunikation Turnier, Match

Der direkte Aufruf ist in drei generelle Abschnitte zu unterteilen. Im Ersten wird eine neue Instanz des Matches erstellt und die benötigten Parameter gesetzt:

```
Match match = new Match("example_matchname");
match.setBot1(Bot1);
match.setBot2(Bot2);
```

Ab diesem Zeitpunkt kann das Match gestartet werden. In diesem Fall benutzt das Match die Standardparameter. Im Anschluss können optionale Parameter gesetzt werden um die Standardparameter zu überschreiben. Im Turnier muss dies einheitlich für jedes Match gleich gesetzt werden.

```
if (setTimeout) match.setTimeout(timeout_turn);
if (setAnotherProperty) match.setAnotherProperty(anotherProperty);
```

Sobald alle Parameter gesetzt sind, kann das Match gestartet werden. Von dem Zeitpunkt an, ab dem es gestartet wurde, bis das Ergebnis abgefragt wurde, laufen die Matches, sowie das Turnier selbst parallel.

```
Furure<String> futureResult = executor.submit(match);
//
//andere Aufrufe parallel mit dem Match
//
String result = futureResult.get();
```

Sobald $futureResult.get()$ aufgerufen wird, blockiert das Programm, bis das Match beendet ist.

Protokoll - Kommunikation Turnier, Match

Dieses Protokoll beschreibt die Kommunikation zwischen dem Turnier und Matches mit std.in und std.out als Schnittstelle.

START <bot1> <bot2> .. <botN> wird vom Turnier an ein neues Match gesendet. Als Argumente werden die Namen der am Match beteiligten Bots mitgegeben. Daraufhin sollte das Matcn sich initialisieren. Im Anschluss antwortet das Match mit OK.

```
Turnier:
    START mcts1 AlphaBetaBot1
Match:
    OK
```

INFO <key> <value> wird vom Turnier an das Match gesendet. Die gesendeten Informationen sollten vom Match gespeichert werden und enthalten Informationen über die Regeln des Turniers. In Tabelle 3.2 sind die existierenden Paare, diese können jedoch erweitert werden.

timeout_turn	- Zeit für einen Zug in ms
space_limit	- Grenze für Arbeitsspeicher in Mb
initial_board	- Dateiname für initiale Brettstellung
match_name	- Name des Matches, genutzt für Benennung von Log-Dateien

Tabelle 3.2: Key-Value Paare für INFO Nachrichten

Im Anschluss antwortet das Match mit OK.

```
Turnier:
    INFO timeout_turn 2000
Match:
    OK
```

BEGIN <bot> Wird vom Turnier an das Match gesendet. Der mitgesendete Name, bestimmt den Bots, welcher den ersten Zug ausführt.Das Match antwortet mit OK und beginnt das Spiel. Im Anschluss wartet das Turnier auf Abschluss des Spiels. Das Match antwortet nach Spielende mit einem FINISH.

FINISH Wird vom Match alleinstehend nach Abschluss des Spiels an das Turnier gesendet. Im Anschluss sendet das Match die Ergebnisse als **<key> <value>** Paar. Der Inhalt ist stark von der Art des Spiels abhängig. Die erhaltenen Werte werden vom Tunrier in einer Tabelle gespeichert und mit den Ergebnissen anderer Spiele verrechnet, daher muss **<value>** eine Zahl enthalten. Nachdem das Ergebnis übermittelt worden ist schließt das Match mit **DONE**. Das Match sollte nach senden dieser Nachricht terminieren.

```
Turnier:
    BEGIN mcts1
Match:
    OK
    FINISH
    mcts 2
    AlphaBetaBot1 -1
    DONE
```

Die Ergebnisse der Runden werden vom und Turnier in einer .csv Datei gespeichert.

3.6.2 Match und Bot

Jedes Match kommuniziert standardmäßig mit genau 2-Bots. Es würde über den Umfang dieser Arbeit hinausgehen, jedoch ließe sich das Protokoll ohne größeren Aufwand auf N-Spieler erweitern. Das Match startet die Bots und initiiert die Kommunikation.

Die Kommunikation erfolgt über die standard Input-und Outputstreams. Aus sicht des Bots wird also Input aus der Kommandozeile gelesen und dorthin die Antwort geschrieben.

INFO <key> <value> wird vom Match jeweils an beide von ihm verwalteten Bots gesendet. Diese Nachrichten enthalten die Informationen, welche das Match vom Turnier erhalten hat. Das Match muss alle vom Turnier durch INFO Nachrichten gesetzte <key> <value> Paare weitergeben. Die existierenden Paare sind in Tabelle 3.2
Nachdem er diese Nachrichten erhält, muss der Bot den Inhalt speichern. Wenn intital_board vom Bot empfangen wird, so sollte dieser die übergebne Datei einlesen und seinen internen Zustand entsprechend anpassen.Nach dem empfangen von match_name sollte der Bot den Namen seiner Log-Datei anpassen. Im Anschluss antwortet der Bot mit OK.
Sollte Der Bot bereits TURN oder START erhalten haben, sollte jede INFO Nachricht ignoriert werden.

```
Match:
    INFO timeout_turn 2000
Bot:
    OK
```

TURN <value> wird vom Match an einen der Bots gesendet. <value> enthält den Zug des vorherigen Spielers. Der Bot antwortet mit <own_value> welcher seinen generierten Zug enthält. Wie <value> und <own_value> aussehen und, ob mehrere Züge aufeinander folgen, ist vom Spiel abhängig. Wenn mehrere Züge aufeinander folgen, dann müssen diese durch etwas anderes als ein Leerzeichen getrennt sein.
Hier einge Vorschläge für die Form des Spielzugs: <x-y;x-y>, <P_x-y>, oder <x-y>. Dabei sind x und y die Koordianten auf einem zweidimensionalen Spielbrett und P eine Repräsentation des Spielsteins. Bei Spielen wie Gomoku, bei denen jeder Spieler nur über eine Art Stein verfügt, kann dieser Teil weggelassen werden.

```
Match:
    TURN 10-10
Bot:
    9-10
```

START wird vom Match an einen der beiden Bots gesendet und signalisiert dem Bot, dass er den Eröffnungszug hat. Der Bot generiert seinen Zug und antwortet im gleichen Schema wie bei TURN.

```
Match:
    START
Bot:
    D_7-10
```

END wird vom Match an die Bots nach Spielabschluss gesendet. Die Bots sollten nach empfangen dieser Nachricht temporäre Daten bereinigen und selbst terminieren.

ERROR <msg> wird vom Bot an das Match im Fall eines Fehlers gesendet. Der Bot wartet daraufhin auf eine neue Nachricht vom Match.

3.6.3 Match und Game

Jedes Match kommuniziert mit einer von ihm erstellten Instanz einer Spielsimulation. Diese Instanz ist vom Interface *Game*. Da Game als ein Objekt von Match existiert, entsteht weniger Overhead, als wenn für jedes Match ein weiterer Prozess gestartet wird.
Dadurch muss das Spiel jedoch in Java implementiert werden, was im Konflikt mit (2) steht. Allerdings ist die Implementation des Spiels von geringem Aufwand ist und schränkt die Studierenden nicht in ihrer eigentlichen Arbeit ein. Auch muss das Spiel nur ein einziges mal Implemetiert werden, wenn eine Arbeit zu einem neuen Spiel geschrieben wird. Daher überwiegt die Erfüllung von (4).

4 Beispielhafte Anwendung mit Gomoku

Um die Benutzung zu veranschaulichen wurde ein Turnier mit Gomoku-Bots durchgeführt, sowie ein bereits vorhandener Ǵomoku-Bot in Python für das Interface angepasst.

4.1 AlphaBetaBot

Hier wird die Implemetation eines Bots für Gomoku vorgestellt. Dieser Bot arbeitet mit AlphaBeta-Suche und benutzt die vorgestellten Vorgaben und Optimierungen.

Der Bot benötigt eine Konfigurationsdatein $config.properties$. Diese liegt, der Dateistruktur entsprechend im gleichen Ordner wie die ausführbare Datei. In Abbildung 4.1 ist der Inhalt einer solchen

```
#main configs
depth=3
shuffle=false
size=15
```

Abbildung 4.1: beispielhafte Konfiguration des Bots

Datei zu erkennen. In diesem Fall wird der Bot mit einer Suchtiefe von 5 arbeiten. Dies liegt daran, das $depth$ die Tiefe der rekursiven Aufrufe bestimmt. Bei $depth = 0$ wird kein rekursiver Aufruf gemacht und ein Suchbaum mit Tiefe 2 durchsucht.
Mit $shuffle$ wird festgelegt, ob die List der möglichen Züge bei der Rekursion gemischt wird. Durch $size$ wird die Dimension des Bretts angegeben. Diese ist hier 15 und sollte sich immer mit der Brettgröße der Spielinstanz decken. Für das Turnier hatten alle Bots $size = 15$. Die weitere Konfiguration ist in Tabelle 4.1 aufgeführt.

Bot	depth	shuffle
Bot3	1	false
Bot3S	1	true
Bot4	2	false
Bot5	3	false

Tabelle 4.1: Konfiguration der Bots

4.2 Turnier

Zum demonstrieren der Schnittstelle wurde ein Turnier mit Gomoku-Bots durchgeführt. Am Turnier haben 4 Bots mit unterschiedlichen Konfigurationen teilgenommen. Die Auswertung wird in gekürzter Form stattfinden, da es sie nicht im Fokus dieser Arbeit liegt. Bei dem Turnier wurde vom Turnier von außen die Zugzeit erfasst und von den Bots Zugzeit, Iterationen und Cutoffs. Eine Iteration repräsentiert hier eine Evaluation eines Knotens im Suchbaum. Ein Cutoff ist ein durchgeführter Alph-,oder Beta-Cutoff.

Das Turnier wurde mit den folgenden Argumenten aufgerufen.

```
Bot4 Bot3 Bot3S Bot5 -t 300000 -g 60000 -n AlphaBeta -m 2
```

Daraus gehen die 4 beteiligten Bots, sowie der Turniername «AlphaBeta», eine Zugzeit von 5 Minuten, eine Toleranz von einer Minute, sowie 2 Spiele für jeden Begegnung zweier Bots.

4.2.1 Erwartete Ergebnisse

Es ist zu erwarten, dass Die Bots mit höherer Tiefe, besser sind. Also am Ende des Turniers höhere Punkte und damit eine höhere Gewinnrate haben. Die Beiden Bots, Bot3 und Bot3S, sollten keinen Unterschied in der Wertung aufweisen, da der Einzige Unterschied in dem Mischen der Zuge liegt. Da Alpha-Beta Suche einen Optimalen Zug liefert, müssten sie auch die gleichen Züge generieren.

4.2.2 Ergebnisse

Die Ergebnisse sind Tabelle 4.2 zu entnehmen. Entgegen der ursprünglichen Annahme hat Bot4 eine höhere Punktzahl, als Bot5. Dies ist mit dem Verlieren von Bot5 durch Timeouts zu erklären. Bot 5 hatte insgesamt 3 Timeouts, einmal gegen jeden der anderen Bots in den folgenden Matches verloren:«2-2-1», «1-2-2», «3-1-2». Die Bots, Bot3 und Bot3S, haben wie erwartet die gleichen Punk-

Bot	Punkte
Bot4	4
Bot5	0
Bot3	-2
Bot3S	-2

Tabelle 4.2: Ergebnisse des Turniers zusammengefasst

te erzielt. In ihrer Begegnung im Turnier haben sie ebenfalls, wie erwartet, die gleichen Spielzüge generiert. Die generierten Züge der beiden Spiele sind in Tabelle 4.3. Der einzige Unterschied zwischen den beiden Runden ist, welcher Bot den Eröffnungszug gespielt hat. Der eröffnende Bot hat die Runde auch gewonnen.
Auch ist in Tabelle 4.3 gut die Latenz der Schnittstelle zu erkennen, die bei Match <3_2_2> bei ca. 1ms liegt. Zum Veranschaulichung des internen Erfassens der Metriken steht in Tabelle 4.4 der vollständige Inhalt einer von einem Bot generierten .csv Datei.

3_2_1			3_2_2				
extern erfasst			extern erfasst			intern erfasste Daten von Bot3	
Bot	Zug	Zeit	Bot	Zug	Zeit	Time	Turn
Bot3S	5-5	1	Bot3	5-5	1		
Bot3	5-4	2	Bot3S	5-4	4		
Bot3S	6-5	5	Bot3	6-5	4	4	6-5
Bot3	4-4	7	Bot3S	4-4	8		
Bot3S	4-5	7	Bot3	4-5	9	8	4-5
Bot3	3-4	13	Bot3S	3-4	9		
Bot3S	7-5	14	Bot3	7-5	14	13	7-5
Bot3	4-3	12	Bot3S	4-3	10		
Bot3S	3-5	19	Bot3	3-5	27	26	3-6

Tabelle 4.3: Generierte Züge der Matches «3_2_1» und «3_2_2»

4.3 Match mit Python Bot

Zum testen der Schnittstelle hat Yamac Eren Ay, sie für den bei seiner Arbeit entstandenen Gomoku Bot implementiert.[2] Dazu wurde das Python-Programm mit pyinstaller[1] als ausführbare Datei zur Verfügung gestellt. Dafür wird der Bot durch eine sparate main aufgerufen, wodurch Kommandozeilen Argumente vermieden werden. Es wurde ein Match zwischen einem AlphaBetaBot mit $depth = 3$ und einem Python Bot mit $ZERO$ Python Bot durchgeführt. Das Match fand auf einem $10x10$ großen Brett statt. In Tabelle 4.5 stehen die von der Schnittstelle von außen erfassten Werte.

Nennenswert hier, war der Fall, dass Game, AlphaBetaBot und ZERO ein unterschiedliches Format für die Board-Datei benötigt haben.

[1] https://pyinstaller.org/en/stable/, 11.07.2001

Turn	Time	Iteration	Cutoffs
6-5	12	3073	275
3-4	54	15801	1025
4-5	140	39153	2066
7-5	231	78764	3816
4-3	459	156313	9254
2-2	745	257746	14208
6-7	1105	386177	20695
2-4	1132	405306	14881
2-5	1284	456829	16039
6-9	1505	537271	17991
6-10	1610	594978	20864
6-11	1404	528926	23993
2-8	1740	652148	30303
2-9	2023	765051	33554
2-10	1938	727057	32382
2-12	2306	865144	37710
2-14	2622	992913	42905
7-2	2229	842014	38945
7-0	2531	964785	40604
7-4	1824	670328	36777
7-6	1545	561151	26541
3-2	1411	498327	30386
7-8	956	352695	15219
7-9	778	279442	13880
3-5	233	73225	18634

Tabelle 4.4: Erfasste Daten von Bot4 in Match <2_1_2>

Bot	Turn	Time
zero	7-7	7
AlphaBetaBot	7-6	78
zero	5-7	3
AlphaBetaBot	8-7	161
zero	4-7	2
AlphaBetaBot	6-7	324
zero	3-7	5
AlphaBetaBot	6-5	554
zero	2-7	5
AlphaBetaBot	1-7	4049
zero	6-8	3
AlphaBetaBot	5-4	5274
zero	4-6	2
AlphaBetaBot	4-3	3617

Tabelle 4.5: Ergebnisse des Matchs zwischen AlphaBetaBot und ZERO

5 Diskussion

Die vorgestellte Schnittstelle erfüllt das Ziel, die Möglichkeit zu schaffen, Bots verschiedener Arbeiten zu vergleichen. Dabei mussten jedoch in bestimmten Bereichen Kompromisse eingegangen werden.

5.1 Erfüllung der funktionalen Anforderungen

Die in 3.1 dargelegten Anforderungen wurden wie folgt erfüllt.

(1)Metrikanwendung
Das System ist in der Lage die häufig verwendeten Metriken, Zugzeit und Gewinnrate, standardmäßig zu erfassen und zu erkennen wenn Zugzeit überschritten wurde. Die erfassten Metriken werden gut lesbar, standardisiert in .csv gespeichert, mit der Möglichkeit die erfassten Metriken zu erweitern.

(2)Geringe Einschränkung für den Nutzer
Die Schnittstelle ermöglicht eine freie Wahl der Programmiersprache für die Bots, unter der Voraussetzung, dass eine nativ ausführbare Datei zur Verfügung gestellt werden kann. Da Schnittstelle führt zu keiner Einschränkung für die Art der Implementation. So wird zum Beispiel keine Thread-Sicherheit gefordert.

(3)Geringer Aufwand für den Nutzer
Der Aufwand bei der Benutzung wurde ist gering wie möglich gehalten. Die Nutzung verlangt kein Tieferes Verständnis bestimmter dritt-Partei Software. Die einzige Anforderung ist die Implementierung des Protokolls in Kapitel 3.6.2, mit Standardinput- und Output, sowie ein zur Verfügung stellen als native ausführbare Datei.

(4)Effizienz
Die Schnittstelle erlaubt paralleles ausführen mehrerer Matches, sodass alle beteiligten Bots potentiell parallel laufen können.

5.2 Limitationen der Arbeit

In der Theorie erlaubt die Schnittstelle eine Kommunikation von Bots vollständig unabhängig der verwendeten Programmiersprache. Im Rahmen dieser Arbeit wurde jedoch ausschließlich Java und Python als Sprache für die Bots verwendet.

Die Arbeit beschränkt sich auf den konkreten Fall von rundenbasierten 2-Spieler Spielen. In der Theorie lässt sich die vorgeschlagene Schnittstelle auch auf n-Spieler Spiele erweitern.

6 Fazit

Ziel war es einen Möglichkeit zu finden, damit bei Bachlorarbeiten im Fachbereich Neurotechnologie entstandene Bots, miteinander verglichen werden können. In der Arbeit wurde eine solche Schnittstelle vorgestellt, die es ermöglicht Bots für rundenbasierte 2-Spieler Spiele gegeneinander zu testen. Dabei wurden neben den Spezifikationen der Schnittstelle auch die relevanten Entscheidungen begründet.

Die Schnittstelle ermöglicht die Integration der Bots durch das Nutzen von ausführbaren Dateien und Std.in und Std.out als universelle Schnittstelle. Auch wurde ein Schema vorgeschlagen, um die erhobenen Daten effektiv miteinander zu verwenden.

Durch Nutzung des Schnittstelle in der Zukunft, können bei neuen Arbeiten, effektiv auf die bereits existierenden Bots zugegriffen werden um diese als Referenz zu nutzen.

7 Lektionen

Im Verlauf der Entwicklung sind ein paar Situationen entstanden, welche initial zu größeren Problemen geführt haben. Da die daraus gelernten Lektionen, an anderer Stelle fehl am Platz sind, ich dennoch das gelernte weitergeben möchte, führe ich diese Situationen hier auf.

Java: aufrufen von close()

Bei der Implementation des AlphaBeta Bot, trat der Fall ein, dass der Std.out der Bot Prozesses geschlossen war. Dadurch war ich nicht mehr in der Lage mit dem Prozess zu kommunizieren. In Java gibt ein Aufruf von $close()$ nicht nur das Objekt selbst frei, sondern gibt auch alle assoziierten Ressourcen wieder frei.

```
outputStream = process.getOutputStream();
writer = new BufferedWriter(new OutputStreamWriter(outputStream));

writer.write("Hello World!");

writer.close();
```

Hier exemplarisch der fehlerhafte Code. Durch die Getter-Methode vom Prozess wird ein Output-Stream zurückgegeben, welche mit dem Std.in des Prozesses assoziiert ist. So wird beim Aufruf von $writer.close()$, der Writer selbst, der OutputStream und der Std.in des Prozesses geschlossen.
Dieses Problem kann gelöst werden indem immer nur ein Objekt mit dem assoziierten Stream interagiert, wie es auch bei der Hilfsklasse BotInstance der Fall ist.

Lauch4J hinterlässt verwaiste Prozesse

Diese Situation entstand beim implementieren von Logging für das Botgerüst. Nach einem Test, bemerkte ich eine unlöschbare .log Datei, welche mehrere hundert Megabyte groß war und weiter wuchs. Nach dem manuellen terminieren aller laufenden Javaprozesse, hörte auch die .log Datei auf zu wachsen. Angenommen ein Java Bot, wurde mit Lauch4J als Wrapper, als eine alleinstehende ausführbare Datei zur Verfügung gestellt.

```
bot.getProcess().destroy();
bot.getProcess().waitFor(timeout, TimeUnit.MILLISECONDS);
bot.getProcess().destroyForcibly();
System.out.println(bot.getProcess().isAlive())
```

Nach dem ausführen dieses Codes wird $false$ ausgegeben. Das heißt, der Prozess welcher von $bot.getProcess()$ zurückgegeben wird, hat terminiert. Der Bot selber wird jedoch weiterlaufen und

weiter .log Dateien schreiben. Bei ausführbaren Dateien, welche durch graalvm native-image generiert wurden, besteht dieses Problem nicht.

Das lässt vermuten, dass beim terminieren des Programms, lediglich der Wrapper, terminiert wird. Der eigentliche Prozess bleibt jedoch als Waise zurück.

Immer while-Schleifen ankern

Beim Testen des CSV-Handlers hat das Programm, wiedere rwarten nicht terminiert. Nach gut 10 Sekunden habe ich es dann manuell terminiert. Daraufhin ist sowohl meine IDE, als auch mein File-Explorer abgestürzt. Beides lies sich auch nicht mehr starten, ohne sofort wieder einzufrieren und abzustürzen. Glücklicherweise konnte ich über das Terminal mit $catfilePath/CSVHandler.java$ den Code angucken.

```
while (outFile.exists()) {
        counter++;
        outFile = new File(filePath+File.separator+filename+
                                counter+".csv");
        outFile.createNewFile(); //fälschlicherweise in der Schleife
    }
    outFile.createNewFile();//eigentlich Richtiger Ort
```

Der Code soll eigentlich über Dateinamen iterieren, bis ein unbenutzter gefunden wurde, und dann eine neue Datei erstellen. Mir ist jedoch der Befehl zum Datei erstellen in die While-Schleife gerutscht, wodurch ca. 700.000 neue .csv Dateien erstellt wurden.
Nachdem ich diese .csv Dateien mittels Comandozeile wieder gelöscht hatte, funktionierte alles wieder.

Daher die Lektion: Bevor nicht gesichert ist, dass die While-Schleife verlassen werden kann, oder sie zumindest ohne Schaden anzurichten läuft, immer einen Überprüfung einbauen, welche die Schleife verlassen kann und einen Fehler produziert.

Literaturverzeichnis

[1] L.V. Allis. *Searching for solutions in games and artificial intelligence*. Rijksuniversiteit Limburg, January 1994.

[2] Yamac Eren Ay. Alphazero gegen mcts: Eine studie zur ki-basierten spielstrategie in gomoku. https://doc.neuro.tu-berlin.de/bachelor/2024-BA-YamacAy.pdf, 2024. Bachelor's Thesis, Technische Universität Berlin.

[3] Semra Ayalp. Vergleich der suchalgorithmen alpha-beta, mtd(f) und best-first-minimax für das othello-spie. https://doc.neuro.tu-berlin.de/bachelor/2024-BA-SemraAyalp.pdf, 2023. Bachelor's Thesis, Technische Universität Berlin.

[4] Katherina Babenkova. Varianten der monte-carlo tree search für das brettspiel gomoku unter berücksichtigung der sudden win/death problematik. https://doc.neuro.tu-berlin.de/bachelor/2023-BA-KatherinaBabenkova.pdf, 2023. Bachelor's Thesis, Technische Universität Berlin.

[5] Cameron Browne. Modern techniques for ancient games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[6] Cameron Browne, Matthew Stephenson, Éric Piette, and Dennis J. N. J. Soemers. A Practical Introduction to the Ludii General Game System. In Tristan Cazenave, Jaap van den Herik, Abdallah Saffidine, and I-Chen Wu, editors, *Advances in Computer Games*, Lecture Notes in Computer Science, pages 167–179, Cham, 2020. Springer International Publishing.

[7] Wolfgang Ertel. Suchen, Spielen und Probleme lösen. In *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*, pages 99–135. Springer Fachmedien, Wiesbaden, 2016.

[8] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns : designing, building, and deploying messaging solutions*. The Addison-Wesley signature series. Addison-Wesley, Boston, Mass. ; Munich u.a., 13. print. edition, 2009.

[9] Petr Laštovička. Gomoku ai protocol. https://plastovicka.github.io/protocl2en.htm. Accessed: 19.02.2024.

[10] Alessio Mossudu. Systematischer vergleich von alpha-beta und monte-carlo tree search für das spiel othello. https://doc.neuro.tu-berlin.de/bachelor/2023-BA-AlessioMossudu.pdf, 2023. Bachelor's Thesis, Technische Universität Berlin.

[11] É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne. Ludii – the ludemic general game system. In G. De Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, editors, *Proceedings of the 24th European Conference*

on Artificial Intelligence (ECAI 2020), volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 411–418. IOS Press, 2020.

[12] Celso C. Ribeiro, Sebastián Urrutia, and Dominique de Werra. Leagues, Tournaments, and Schedules. In Celso C. Ribeiro, Sebastián Urrutia, and Dominique de Werra, editors, *Combinatorial Models for Scheduling Sports Tournaments*, pages 1–20. Springer International Publishing, Cham, 2023.

[13] Kuan Liang Tan, Chin Hiong Tan, Kay Chen Tan, and Arthur Tay. Adaptive game AI for Gomoku. In *2009 4th International Conference on Autonomous Robots and Agents*, pages 507–512, February 2009.

[14] Gün Yanik. Heuristische vs. stochastische suche - alphabeta pruning und monte carlo tree search im spiel der amazonen. https://doc.neuro.tu-berlin.de/bachelor/2021-BA-GuenYanik.pdf, 2021. Bachelor's Thesis, Technische Universität Berlin.