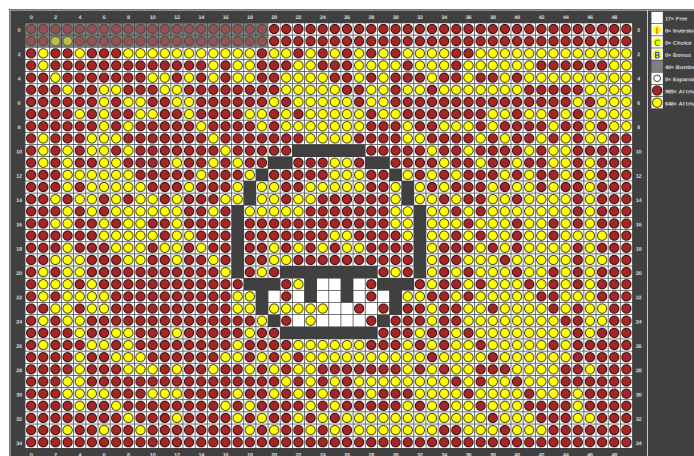


**Fakultät
Informatik und Mathematik**

Projektbericht

zum Wahlpflichtfach im SS 2019

Projekt: Client-K.I.s (ReversiXT)



Gruppe: 12

Autoren: matthias.goetz@st.oth-regensburg.de
simon.hofmeister@st.oth-regensburg.de
dominik1.meier@st.oth-regensburg.de

Leiter: Prof. Dr. rer. nat. Carsten Kern

Abgabedatum: 05.07.2017

Inhaltsverzeichnis

1	Einleitung	1
2	Allgemeine Informationen	3
2.1	Team und Kommunikation	3
2.2	Aufgabenverteilung	3
2.3	Technische Daten	4
2.4	Datenstruktur	6
3	Heuristiken	8
3.1	Bedeutung der Heuristiken	8
3.2	Heuristik zur Bewertung von Randpunkten	8
3.3	Heuristik zur Bewertung von Spielern	9
3.4	Verwendete Heuristik	10
4	Statistiken	12
4.1	Vergleich "Paranoid", AlphaBeta-Pruning, AlphaBeta-Pruning mit Zugsortierung	12
4.2	Verbesserung durch Zugsortierung	13
4.3	Vergleich ohne und mit Iterative Deepening	15
4.4	Vergleich ohne und mit Aspiration-Windows	16
4.5	Parametrierung	17
5	Bombenphase	20
6	Wettbewerbs-Spielfelder	23
7	Logging	26
8	GUI	27
9	Profiling	28
10	Fazit	30
	Anhang	I
A	Abbildungsverzeichnis	I

1 Einleitung

In diesem Projekt entwickeln wir im Rahmen des Kurses "Client-K.I.s (ReversiXT)" der OTH-Regensburg unsere erste künstliche Intelligenz (abgekürzt: KI), die fähig sein wird, das Spiel Reversi (auch: Othello) zu spielen. Bei der Grundvariante des Spiels spielen 2 Spieler auf einem Spielbrett mit 8x8 Feldern rundenbasiert gegeneinander. Zu Beginn hat jeder Spieler 2 Steine, die wie folgt angeordnet sind:

Nun beginnt der erste Spieler seinen Zug. Folgende Züge sind gültig:

- Das Feld auf das ein Stein gesetzt wird ist noch nicht belegt.
- Man muss ein oder mehrere gegnerische Steine zwischen einem eigenen und dem neu gesetzten Stein einschließen.
- Zwischen den eigenen Steinen darf kein leeres Feld sein.

Jeder Spielstein der eingeschlossen wurde, wird durch einen Stein der eigenen Farbe ersetzt.

Um der KI eine Herausforderung zu bieten, wurden das Spiel wie folgt abgeändert:

- Das Spielfeld hat eine variable Größe und bis zu 50x50 Felder.
- Auf dem Spielfeld spielen bis zu 8 Spieler gegeneinander.
- Es existieren Transitionen (Erklärung folgt später).
- Expansionssteine 'x' verhalten sich wie gegnerische Steine eines disqualifizierten Spielers (die Steine bleiben auf dem Spielfeld, jedoch kann von diesen kein Zug ausgeführt werden).
- Jeder Spieler erhält eine kartenspezifische Anzahl Überschreibsteine, mit denen man bereits besetzte Spielfelder durch den eigenen Stein ersetzen darf. Dies ist jedoch nur möglich, wenn dadurch mindestens ein gegnerischer Stein eingeschlossen wird, oder das überschriebene Feld ein 'x' war (Expansionsstein-Sonderregel).
- Wenn kein Spieler mehr ziehen kann, kann jeder Spieler eine kartenspezifische Anzahl an Bomben werfen, die je nach Stärke, alle getroffenen Spielfelder aus dem Spiel entfernt.
- Wir implementieren Sonderfelder, auf die man setzen kann: Choice, Inversion und Bonus. Bei Choice kann man mit einem beliebigen Spieler die Steine tauschen, bei Inversion werden die Steine reihum durch rotiert. Das Bonusfeld erlaubt die Auswahl eines zusätzlichen Überschreibsteins oder einer Bombe. Diese Felder können nicht eingeschlossen werden.
- Jeder Spielzug besitzt ein Zeitlimit von beispielsweise 1 Sekunde.

Das Programm soll in der Lage sein, eine ASCII-codierte Karte von einem Linuxserver via einer Socket-Verbindung entgegen zu nehmen und auf dieser entsprechend zu spielen, sowie eine getroffene Entscheidung für einen Spielzug an den Server weiter zu leiten. Multithreading wurde aus Fairness untersagt, so dass jede Gruppe aus Studenten die gleiche Chance besitzt, die beste KI des Kurses zu entwickeln.

Der allgemeine Ablauf des Spiels sieht aus der Sicht eines Clients schematisch aus wie in Abb. 1 angedeutet:

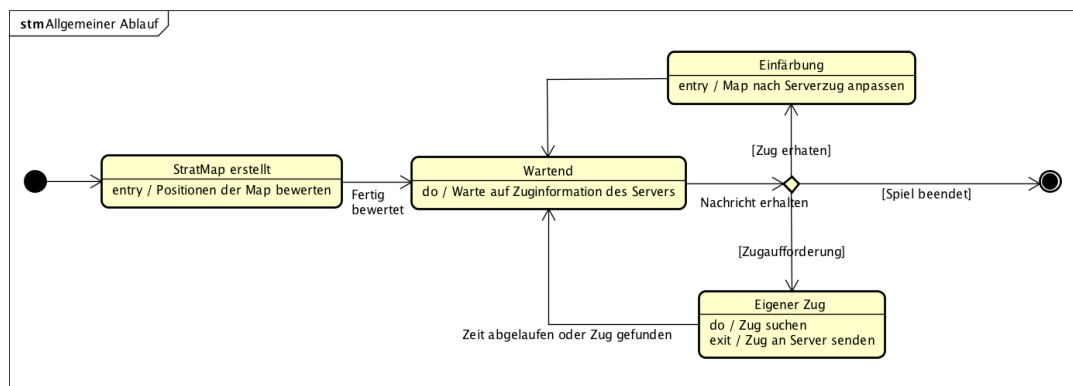


Abbildung 1: Allgemeiner Ablauf

2 Allgemeine Informationen

2.1 Team und Kommunikation

Unser Team besteht aus den 3 Autoren dieses Berichts:

Dominik Meier, Student der Technischen Informatik im 4.Semester,

Matthias Götz, Student der Wirtschaftsinformatik im 4.Semester,

Simon Hofmeister, Student der Allgemeinen Informatik im 3.Semester.

Zusätzlich zu unserem regelmäßigen Vorlesungen und Übungsstunden vereinbarten wir regelmäßige persönliche Treffen zur Diskussion von Algorithmen und Datenstrukturen sowie bei der Effizienz. Auch konkrete Umsetzungen, Aufgabenverteilungen und gegenseitige Motivation waren Gegenstand der Treffen. Folgende Kommunikationsmedien wurden genutzt:

Kommunikationssystem	Art der Mitteilungen
SSE-Lab-Mailing-Liste	Wichtige grundlegende Entscheidungen und permanent wichtige Informationen, Meilensteine
Whatsapp	Statusupdates, Algorithmendiskussion
Discord	Aufgabenverteilung, Teambuilding, Diskussion von Algorithmen und Programmierstandards

Tabelle 1: Kommunikationsmittel

2.2 Aufgabenverteilung

Um ein bestmögliches Ergebnis unserer KI zu erreichen, haben wir die Aufgaben so aufgeteilt, dass jeder einen ähnlichen Arbeitsaufwand zu bewältigen hat. Dann wurden die Aufgabengebiete je nach Kompetenz und Interesse verteilt. Hierbei ergaben sich oft weitere Einfälle, wie z.B. die Logging-Funktion oder die GUI, die für das Projekt nicht obligatorisch sind, die wir aber aus Eigeninteresse noch implementiert haben. Leider konnten viele interessante Gedanken aus Zeitgründen nicht mehr umgesetzt werden. Die letztendliche Verteilung der einzelnen Aufgaben findet sich in der folgenden Tabelle.

Aufgabe	Personen
Alpha-Beta-Pruning	Simon
Anpassung von internen Parametern	Dominik, Simon
Aspiration-Windows	Simon
Bewertung Bomben, Überschreibsteine	Matthias
Bombenphase	Dominik, Simon
Einlesen des Spielfelds aus einer Textdatei	Simon
GUI-Programmierung zum ansehen oder mitspielen	Simon
Kommandozeilen-Parameter	Dominik
Logging-Funktion der Spielzüge	Dominik, Simon
Paranoid-Algorithmus	Dominik, Simon
Playback-Analyse	Matthias
Profiler-Analyse	Simon
Server-Kommunikation	Dominik
Spielfeldbewertung	Matthias
Spielerbewertung	Matthias
Start-Skripts Server und Triviale KI	Dominik
Steine einfärben	Dominik, Simon
Wettbewerbsspielfelder	Matthias, Dominik, Simon
Zugfindung	Dominik, Simon
Zugsortierung	Matthias, Simon

Tabelle 2: Aufgabenverteilung

2.3 Technische Daten

Aufgrund dessen, dass wir Studenten sind, haben wir keine einheitlichen Entwicklungssysteme. Daher halten wir hier unsere individuellen Systeme fest, um das Verhalten der KI auf den verschiedenen Systemen besser nachvollziehen zu können.

Person	System	relevante Hardware/-Software	Spezifizierung
Simon	Desktop	Hardware	CPU: Intel i7-6770k
Simon	Desktop	Hardware	RAM: 2x16GB, DDR4-2133
Simon	Desktop	Software	OS: Windows 10 Education, Version 10.0.15063 Build 15063
Simon	Mobile	Hardware	Microsoft Surface 4 Pro
Simon	Mobile	Hardware	CPU: Intel i5-6300U
Simon	Mobile	Hardware	RAM: 4GB DDR3L-1600
Simon	Mobile	Software	OS: Windows 10 Pro, Version 1607 Build 14393.1198
Simon	Mobile	Software	IDE: Eclipse Version: Neon.3 Release (4.6.3) Build id: 20170314-1500
Simon	Desktop	Software	Latex: TexStudio 2.12.4
Simon	Desktop	Software	Latex: MikTex 2.9
Simon	Beide	Software	IDE: Eclipse Version: Neon.3 Release (4.6.3) Build id: 20170314-1500
Simon	Beide	Software	Sprache: Java 8 Update 131 (64-bit)
Simon	Beide	Software	Testing: JUnit Release 4.12
Simon	Desktop	Software	Profiler: YourKit Java Profiler 2016.02 Build 46 (TestVersion 15-Days, 04.06.2017-19.06.2017)
Simon	Beide	Software	SVN: Tortoise SVN 2.4.0.2 64bit
Simon	Beide	Software	Git Version 2.8.1
Matthias	Desktop	Hardware	CPU: AMD FX-8120
Matthias	Desktop	Hardware	RAM: 16GB, DDR3-1600
Matthias	Desktop	Software	OS: Windows 10 Education, Version 10.0.15063 Build 14393
Matthias	Desktop	Software	Latex: TexStudio 2.12.4
Matthias	Desktop	Software	Latex: MikTex 2.9
Matthias	Desktop	Software	Sprache: Java 8 Update 131 (64-bit)
Matthias	Desktop	Software	IDE: Eclipse Version: Neon.3 Release (4.6.3) Build id: 20170314-1500
Dominik	Mobile	Hardware	CPU: Intel i5-6300HQ
Dominik	Mobile	Hardware	RAM: 8GB, DDR4-2133
Dominik	Mobile	Software	OS: Windows 10 Home, Version 1607 Build 14393.1066
Dominik	Mobile	Software	Latex: TexStudio 2.12.4
Dominik	Mobile	Software	Latex: MikTex 2.9
Dominik	Mobile	Software	Sprache: Java 8 Update 132 (64-bit)
Dominik	Mobile	Software	IDE: Eclipse Version: Neon.3 Release (4.6.3) Build id: 20170314-1500

Tabelle 3: Entwicklungsbedingungen Hard- & Software

2.4 Datenstruktur

Die originale Karte besteht aus einzelnen ASCII-Zeichen die per Leerzeichen oder dem newline-Zeichen getrennt sind. Unsere Entscheidung wie wir das Spielbrett speichern, fiel auf einen 2-dimensionalen char-Array, da das Ein- und Auslesen einfach funktioniert und einen sehr schnellen Zugriff bietet, zudem ist das Spielfeld von der Größe her statisch fest, eine andere Datenstruktur macht aus unserer Sicht wenig Sinn. Die einzelnen Felder als Objekte zu speichern macht bei der maximalen Anzahl an Feldern auf einem Spielfeld mit $50 \times 50 \rightarrow 2500$ Spielfeldern keinen Sinn, da hier zu viel Zeit verbraucht wird auf jedes Objekt zuzugreifen, den Wert auszulesen und dann eventuell zu verändern.

Die Überschreibsteine, Bomben und Anzahl der noch existierenden Spieler speichern wir aus den selben Gründen in eindimensionalen Arrays (int bzw. boolean).

Etwas mehr Schwierigkeiten hatten wir bei der Auswahl einer geeigneten Datenstruktur für die Transitionen. Hier haben wir eine variable Anzahl, die wir nicht vorher berechnen oder auslesen können. Daher fallen Arrays auch aus dem Konzept. Transitionen per Baum abzuspeichern, macht aus unserer Sicht auch nur wenig Sinn. Daher waren wir bei dem Thema Collections. Die einzigen für uns in Frage kommenden Collections waren ArrayLists und HashMaps. Dabei sind wir gezwungen, die Transitionen als String abzuspeichern. Bei ArrayLists müssten wir jedes Mal über die gesamte Liste iterieren, dies schien uns sehr aufwändig. HashMaps bieten den Vorteil, dass sie über den Key die Position des Values im Speicher berechnen. Somit haben wir hier einen annehmbar schnellen Zugriff. Zudem existiert mit der "containsKeyMethode eine schnelle Prüfmethode ob an der Koordinate mit der angegebenen Richtung eine Transition existiert. Das Zeitaufwändige ist hier, dass die Koordinaten mit Richtung jedes Mal in einen String umgewandelt werden müssen, wenn man auf eine Transition prüfen will, zudem muss bei Erfolg, der Ergebnis-String wieder in die einzelnen Values aufgeteilt werden. Um nicht die ganze Map mit Key \rightarrow Value durchsuchen zu müssen ob die Transition noch in anderer Richtung vorkommt, speichern wir von Anfang an Key \rightarrow Value und Value(als Key) \rightarrow Key(als Value) ab. Z.B. Transition $x\ y\ z \leftrightarrow x'\ y'\ z'$: $5\ 3\ 1 \leftrightarrow 7\ 2\ 2$ wird bei uns abgespeichert als:

- `hashmap.put("x,y,z", "x',y',z'")` also `hashmap.put("5,3,1", "7,2,2")`
- `hashmap.put("x',y',z'", "x,y,z")` also `hashmap.put("7,2,2", "5,3,1")`

Hier verbrauchen wir doppelten Speicherplatz, den wir in Kauf nehmen, da die Transitionen sowieso verhältnismäßig sehr wenig Speicher verbrauchen und wir nicht unter Speicherknappheit leiden. Der Zeitgewinn ist es uns an dieser Stelle wert.

Die möglichen Züge einer Spielrunde speichern wir in einer dynamischen ArrayList, da nicht bekannt ist, wie viele möglichen Züge zur Verfügung stehen und wir auf sehr viele Elemente

nacheinander zugreifen müssen. Zudem gibt es für ArrayListen bereits sehr gute Sortiermethoden, die uns vielversprechende Methoden möglichst weit nach vorne sortieren kann (mit eigenem Comparable), so dass wir zusätzlich mit dem Alpha-Beta-Pruning sehr viel Zeit gewinnen können.

3 Heuristiken

3.1 Bedeutung der Heuristiken

Heuristiken haben für unsere künstliche Intelligenz einen sehr hohen Stellenwert, da es notwendig ist, den aktuellen Spielstand auszuwerten und dementsprechend zu entscheiden, welches die bestmögliche Aktion ist die durchgeführt werden kann. Hierbei wird unterschieden zwischen Sondersteinen und dem Setzen von Steinen. Dabei hat es eine hohe Bedeutung effiziente und logische Heuristiken zu verwenden, um einen Vorteil gegenüber anderen künstlichen Intelligenzen zu erlangen und bestenfalls das Spiel zu gewinnen. Nach unserer Einschätzung gibt es einige Bewertungsmöglichkeiten, die berücksichtigt werden können, um eine gute KI zu programmieren. Besonders wichtig sind die besondere Bedeutung von Eckpunkten, aktuelle Position des bzw. der gegnerischen Spielern.

3.2 Heuristik zur Bewertung von Randpunkten

Notwendig für diese Heuristik ist die Definition von Randpunkten. Randpunkte sind Punkte im Feld, die von Nachbarn, also anliegenden Punkten durchlaufen werden und der nachfolgende Punkt ein Teil der Spielfeldbegrenzung ist, welches keine Transition hat. Insgesamt müssen 4 verschiedene Durchgänge überprüft werden um alle Nachbarpunkte zu betrachten.

Nach der Bewertung jedes Punktes innerhalb der Spielkarte, mit Ausnahme der Punkte mit dem Wert 0 erhalten, ergibt sich eine strategische Karte in Form eines zweidimensionalen Arrays, das Punkten in dem Feld eine unterschiedliche Bedeutung zuweist, welche genau so groß ist wie die Spielkarte.

z.B:

```

- - -
0 1 -
- - -

```

Betrachtet man in diesem Fall den Punkt mit dem Wert 1, wenn keine Transitionen gegeben sind, besitzt dieser Punkt 0 von 4 möglichen Durchgängen.

Je weniger Durchgänge ein Punkt hat, desto besser ist dieser Punkt und desto höher wird die Bewertung. Ein Punkt der 0 von 4 möglichen Durchgängen hat, kann nicht durch setzen von Steinen eines anderen Spielers eingenommen werden.

Als Bewertung für die Randpunkte ergibt sich folgende Tabelle:

0 von 4 Durchgänge $\hat{=}$ Bewertung 50
 1 von 4 Durchgänge $\hat{=}$ Bewertung -5
 2 von 4 Durchgänge $\hat{=}$ Bewertung 3
 3 von 4 Durchgänge $\hat{=}$ Bewertung 2

4 von 4 Durchgänge $\hat{=}$ Bewertung 1

Ein Vorteil dieser Methode ist, dass sie nach dem Erhalt der Spielkarte nur einmal durchgerechnet werden muss und sich die Bedeutung nicht während des Spiels ändert. Dementsprechend benötigt diese Heuristik eine geringe Rechenzeit und bietet die Möglichkeit jede Position einzeln zu überprüfen.

Ein Nachteil ist, dass man gegebenenfalls einen Punkt bevorzugt der eine höhere Bewertung hat, als mehrere andere Punkte, was auch zum Nachteil werden könnte. Ein weiterer negativer Punkt ist, dass sich die Bewertung nicht während des Spiels anpasst. Zusätzlich sind Punkte mit 1 von 4 Durchgängen am Ende des Spiels auch sinnvoll um die meisten Steine zu besitzen.

Beispiel:

```
- - - - - → 0 0 0 0 0  
- 0 1 2 - → 0 50 -5 50 0  
- - - - - → 0 0 0 0 0
```

3.3 Heuristik zur Bewertung von Spielern

Eine weitere Heuristik ist die Bewertung von Spielern, welche die Spieler bewerten soll, nach den Punkten mit deren zugeteilten Nummer. Diese Heuristik zählt die Punkte, welche jeder Spieler besitzt, oder zählt die Werte aus der ersten Heuristik zusammen. Daraus ergibt sich eine Bewertung der einzelnen Spieler abfallend geordnet nach dem Wert mit der höchsten Zahl. Um eine möglichst gute Platzierung zu erhalten, ist es sinnvoll, den Spielern Steine zu nehmen, die in der Bewertung am nächsten liegen. Je weiter ein Gegner weg von der eigenen Platzierung ist, desto weniger interessant ist es, dessen Punkte einzunehmen.

z.B (Aus Sicht des Spieler 3)

Spieler 4 → 94

Spieler 1 → 74

Spieler 3 → 60

Spieler 2 → 50

Spieler 3 sollte demnach eher die Punkte von Spieler 1 und Spieler 2 nehmen, da diese einen geringeren Abstand zu der eigenen Bewertung haben, um den Abstand zu dem Spieler vor sich zu verringern und den Abstand zu dem Spieler hinter sich zu erhöhen.

Da es maximal 8 Spieler gibt, kann man für die Bewertung 8 abzüglich des Abstandes in der Tabelle zur eigenen Position wählen. Bei Gleichstand der Punkte wählt man jeweils die Bewertung des näheren Spielers.

In dem Beispiel aus Sicht des Spieler 3 wäre die Bewertung so:

Spieler 1 $\rightarrow (8 - 1) = 7$

Spieler 2 $\rightarrow (8 - 1) = 7$

Spieler 4 $\rightarrow (8 - 2) = 6$

Vorteil dieser Heuristik ist es, dass sie bei jedem eigenen Zug die gegnerischen Spieler bewertet und dementsprechend bevorzugt Punkte von Spielern nimmt, die nahe an der eigenen Position sind. Ein Nachteil ist, dass diese Bewertung erst Sinn macht, wenn mehr als 2 Spieler gegeneinander spielen.

3.4 Verwendete Heuristik

Die tatsächlich verwendete Heuristik ist in mehrere Teilfunktionen einzuteilen. Dabei haben wir verschiedene Bewertungen für die 2 Phasen des Spiels. An dieser Stelle wird die erste Phase genauer erklärt, da die Bombenphase später im Projektbericht genau beschrieben wird.

Zur besseren Verständnis kann man über die folgende Tabelle alle notwendigen Werte ablesen.

Nachdem die KI die Map des Servers erhalten hat und eine Zugaufforderung erhält beginnt die Bewertung der strategicMap. Hierbei wird an jeder Stelle die nicht '-' ist überprüft wie viele Durchgänge in alle vier möglichen Richtungen vorhanden sind. Es gibt einen Durchgang in eine Richtung, wenn ein Nachbar der betrachteten Stelle und deren von dem Beobachtungspunkt gegenüberliegende Stelle keine '-' sind falls in diese Richtung keine Transition vorhanden ist. Echte Eckpunkte, also Punkte die keinen Durchgang haben erhalten eine Bewertung indem in alle Richtungen gezählt wird, wie viele Steine maximal eingenommen werden können, da diese ausschließlich mit Überschreibsteinen von anderen Spielern übernommen werden können. Dieser Wert wird dann mit 4 multipliziert. Anliegende Feldern wird der halbe Wert der Eckpunkte abgezogen, da es gefährlich ist an Stellen vor Eckpunkten zu ziehen. Bei Punkten mit nur einem Durchgang werden die Steine in alle Richtungen gezählt, welche keinen Durchgang haben

Bezeichnung	Wert-vergabe	Häufigkeit der Wertveränderung	Berechnung des Wertes
strategicMap[][]	Dynamisch	Einmalig	Positionsbewertung jeder Stelle die erreichbar ist
specialStonesMap[][]	Dynamisch	Jede Runde	Bewertung von Stellen neben Spezialsteinen
playerRank[x][0]	Dynamisch	Jede Runde	Position über den Heuristik-Wert für Spieler x
playerRank[x][1]	Dynamisch	Jede Runde	Heuristik-Wert des x-ten Spielers auf der Map
specialStones[0]	Dynamisch	Zu Beginn und bei Disqualifikationen	$((\text{Bombenstärke} * 2 + 1) * (\text{Bombenstärke} * 2 + 1)) * 2$
specialStones[1]	Dynamisch	Zu Beginn und bei Disqualifikationen	$((\text{Maphöhe} * \text{Mapbreite}) / (\text{Maphöhe} + \text{Mapbreite})) * 5$
HEURISTICPLAYERDISTANCE	Statisch	Einmalig	Wert 16

Tabelle 4: Werte für die Heuristik

und erhalten davon den drittel der maximal einnehmbaren Stellen als Heuristik-Wert addiert. Positionen mit je 2,3,4 Durchgängen wird der Wert von 3,2,1 hinzugefügt.

Nach der Bewertung aller verwendbaren Positionen werden die Spezialsteine Bomben und Überschreibsteine bewertet. specialStones[0] hat den Wert für Bomben und specialStones[1] hat den Wert für Überschreibsteine. Falls mehr als zwei Spieler teilnehmen wird dieser Wert mit dem HEURISTICPLAYERDISTANCE multipliziert, damit das Verhältnis korrekt bleibt. Falls zu einem späteren Zeitpunkt des Spiels nur noch 2 Spieler aktiv sind wird dieser Multiplikator nicht mehr verwendet.

Der Rang von den Spielern wird jede Runde neu berechnet, falls es mehr als zwei aktive Spieler gibt. Dieser Rang ist von Bedeutung bei der Bewertung eines möglichen Zuges. Punkte, die eingenommen werden und keinem Spieler gehören haben die strategicMap Bewertung der Stelle mal HEURISTICPLAYERDISTANCE. Stellen die von anderen Spielern eingenommen geben die Bewertung von strategicMap mal (HEURISTICPLAYERDISTANCE-Positionsabstand zu dem eigenen Platz). Der Abstand lässt sich durch playerRank[x][0], wobei das x für die Spielernummer steht. In der playerRank[x][1] finden sich die jeweiligen Bewertungen der Spieler. Daraus ergibt sich die Gesamtbewertung eines Zuges.

4 Statistiken

Da wir im Projekt nach und nach Optimierungen getroffen haben, ist es natürlich sehr interessant, wie stark sich welche Optimierung auf unseren Algorithmus ausgewirkt hat. Hierbei stehen folgende Punkte im Vordergrund: Wie viel Zeit haben wir weniger gebraucht? Wie groß ist der Unterschied der Steinanzahl zu den Gegnern? Dazu mussten wir erstmal fixe Parameter festlegen um eine Vergleichbarkeit zu erzielen. Feste Suchtiefe, ein fester Gegner (triviale KI), Nutzung der selben Heuristik, Nutzung derselben Hardware für die jeweiligen Vergleiche. Die Vergleiche sind von der Heuristik und den Algorithmen nur in sich geschlossen aussagekräftig, da sich das Programm bis zur Erstellung der weiteren Vergleiche stetig weiterentwickelt hat. Bei den angegebenen Zeiten wurde angenommen, dass die "triviale AI" keine Zeit verbraucht, da deren benötigte Zeit unter den getesteten Bedingungen als einigermaßen konstant angesehen werden kann und uns deren verbrauchte Zeit nicht bekannt ist. Bei den gesonderten Betrachtungen der Zugsortierung, des Iterative Deepening und der Aspiration Windows war das Programm bereits so weit, dass alle Features implementiert waren, daher ist der rechte Teil der jeweiligen Tabellen (mit dem Feature) identisch, während sich die linke Seite (ohne dem Feature) unterscheidet. Zur Simulation des Programms ohne dem Feature, wurde dieses einfach per Kommentarfunktion aus dem Programm entfernt. Getestet wurde auf Simon's Surface Pro 4.

4.1 Vergleich "Paranoid", AlphaBeta-Pruning, AlphaBeta-Pruning mit Zugsortierung

Folgende Zusatzannahme: Feste Map: castle.txt um eine Vergleichbarkeit herzustellen. Hier geht es überwiegend um die Zeitersparnis.

Aus der Tabelle aus Abbildung 2 kann man ablesen, dass die mittlere und maximale Zugzeit vom Paranoid durch das Alpha-Beta-Pruning erheblich besser geworden sind. Bei großen Suchtiefen hat die KI mit AlphaBeta-Pruning jedoch oft verloren, da die Züge nicht nach Wert sondern nur nach Setzposition auf der Karte von links oben nach rechts unten sortiert sind. Hier werden viele gute Züge zu früh abgeschnitten. Dies hat sich durch den Einbau des Standard-MergeSorts für Listen in Java mittels eines Comparable deutlich verbessert. Dies erkennt man im Vergleich der jeweils letzten Spalte. Bei Suchtiefe 5 haben wir statt einem Sieg nun eine Niederlage eingefahren, jedoch haben wir im Schnitt öfter gewonnen als zuvor. Das Ziel ist unter möglichst vielen verschiedenen Umständen gut zu spielen und nicht eine auf einen Fall spezialisierte KI zu bauen, dieses Ziel wurde hiermit gut umgesetzt, auch wenn dies nicht mehr ganz vergleichbar ist, denn die Spiele sind aufgrund der unterschiedlichen Algorithmen unterschiedlich abgelaufen, siehe Suchtiefen sechs und höher. Mit der Beschränkung der KI auf eine

Daten:	Hardware:	Microsoft Surface Pro																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												</
--------	-----------	-----------------------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

Abbildung 2: Vergleich Paranoid, AlphaBeta und AlphaBeta mit Zugsortierung

Sekunde Berechnungszeit haben wir immer noch ein annehmbares Ergebnis erzielt, denn es korreliert von Sieg/Niederlage mit dem Spiel auf Tiefe. Somit sind nicht zu viele relevante Züge abgeschnitten worden.

4.2 Verbesserung durch Zugsortierung

Nun wird die Zugsortierung auf zehn verschiedenen Maps gesondert betrachtet, um besser erkennen zu können, welchen Nutzen man aus der Zugsortierung ziehen kann. In der Theorie hat die Zugsortierung den folgenden Sinn: Die Züge werden nach eigens definierten Kriterien sortiert um die voraussichtlich wertvollsten Züge zu Beginn zu betrachten und diese nicht durch alpha-beta-Pruning abzuschneiden. Zudem wird im Idealfall schneller ein guter Zug gefunden (da die guten Züge zuerst betrachtet werden) wodurch mehr Äste des Suchbaums abgeschnitten werden können. Durch eine erhöhte Restzeit, sollte somit auch eine größere durchschnittliche Tiefe ergeben.

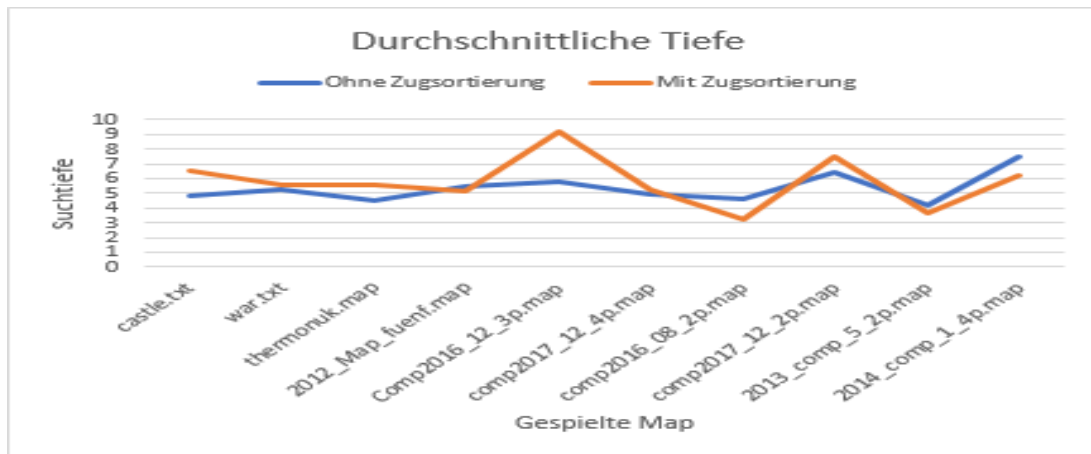


Abbildung 3: Vergleich der durchschnittlichen Suchtiefe mit und ohne Zugsortierung

Wie in Abb. 3 erkennbar ist, erreicht unsere KI auf 4 Maps eine größere, auf 2 Maps eine kleinere und auf 4 Maps eine vergleichbare durchschnittliche Suchtiefe durch die Zugsortierung. Die zwei Maps auf denen wir die Suchtiefe nicht mehr erreichen sind ein Beispiel dafür, dass die Züge, die man von der Heuristik als beste einstufen würde, langfristig gesehen nicht die besten Züge sein müssen. Die Folge ist nun, dass die KI länger suchen muss um diesen Zug zu finden als zuvor, da der Zug ein niedriges Potential hatte und somit ans Ende der möglichen Züge sortiert wurde. Langfristig gesehen war der Zug aber von höherem Nutzen.

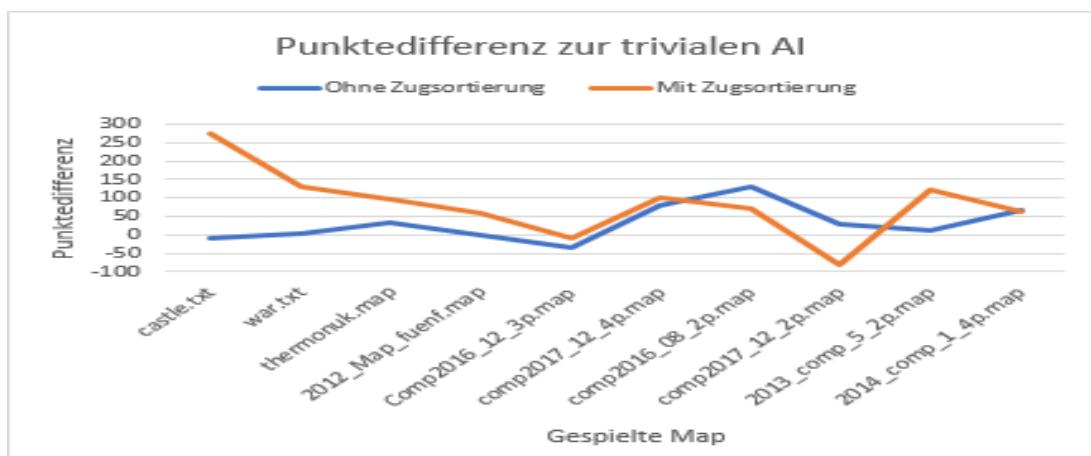


Abbildung 4: Vergleich der Punktedifferenz zur trivialen AI mit und ohne Zugsortierung

Nach Abb. 4 hat unsere KI auf acht von zehn Maps mit Zugsortierung somit mehr oder gleich viele Punkte gegen die triviale AI geholt als zuvor. Auf zwei Maps haben wir mit der Zugsortierung schlechter gespielt als zuvor ohne Zugsortierung. Da das bestreben ist, auf möglichst vielen Karten gut zu spielen, ist eindeutig, welchen Nutzen die Zugsortierung bringt. Die zugehörige Tabelle mit den Daten ist Abb. 14.

4.3 Vergleich ohne und mit Iterative Deepening

Iterative Deepening bedeutet, dass wir nicht gleich die tiefst mögliche Suchebene betrachten, da wir bei Spielen auf Zeit ansonsten mit einem Zug nicht fertig werden und keinen sinnvollen Spielzug finden könnten, da die Verzweigungsbäume auf großer Suchtiefe sehr viele Blätter haben, die überprüft werden müssten. Dies kostet viel Zeit. Mit dem Iterative Deepening beginnen wir bei Suchtiefe 0 (nur unsere möglichen Züge) die wir in der Zeit sicher komplett durchgehen können. Hier erhalten wir einen gültigen Zug. Nun gehen wir zwei Schritte weiter nach unten. Hier erhalten wir einen weiteren gültigen Zug, den wir uns merken können. Sollten wir hier aus Zeitgründen keinen gültigen Zug finden, haben wir ja bereits aus der vorherigen Suche einen Zug den wir spielen können.

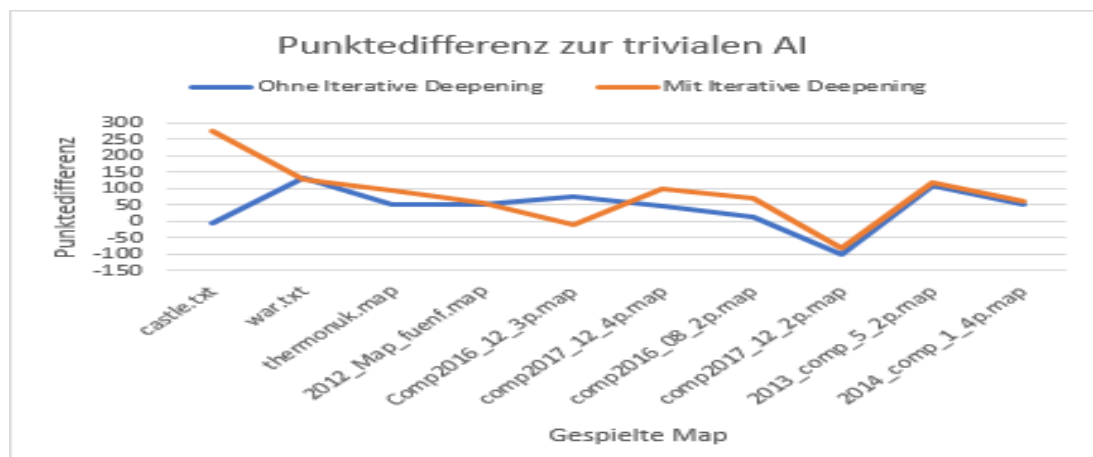


Abbildung 5: Vergleich der Punktedifferenz zur trivialen AI mit und ohne Iterative Deepening

Die Verbesserung der Spielleistung durch das Iterative Deepening wird in Abb. 5 dargestellt. Hierbei erkennt man, dass wir nur auf einer Spielkarte etwas schlechter spielen als ohne Iterative Deepening. Die Verbesserung ergibt sich dadurch, dass wir einen Zug nur dann als bestmöglich betrachten, wenn der komplette Verzweigungsbaum auf dieser Iterationsebene vollständig betrachtet wurde. Um sicherzustellen, dass wir den Baum auf der ganzen Tiefe betrachten können, müssen wir iterativ vorgehen und erst eine sichere Basis aufbauen um sicher einen Zug zu finden, den wir an den Server abschicken können, falls uns die Rundenzeit ausgeht.

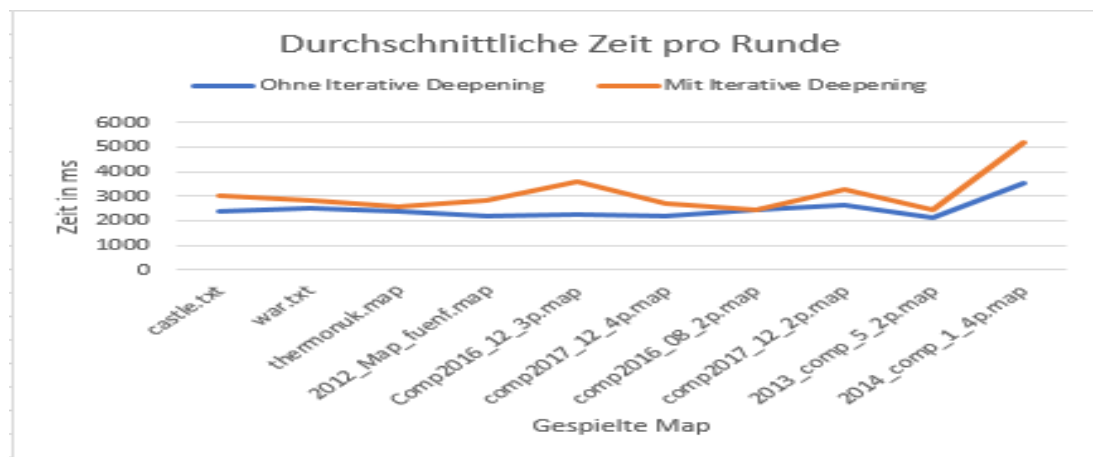


Abbildung 6: Vergleich der durchschnittlich benötigten Zeit pro Spielzug mit und ohne Iterative Deepening

Die Entwicklung der benötigten Zeit pro Runde steigt durch das Feature leicht an, da der Suchbaum bei jeder Iteration neu aufgebaut werden muss. Dies zeigt der Graph in Abb. 6. Beide Graphen beziehen sich auf die zugehörige Tabelle in Abb. 13, jedoch kostet uns dies nur etwa zehn Prozent der Zeit, weswegen wir diese Zeitkosten aufgrund von sicherer Spielweise gerne in Kauf nehmen.

4.4 Vergleich ohne und mit Aspiration-Windows

Die Aspiration-Windows können einen großen Vorteil bringen, denn man engt den Suchbereich der besten Möglichkeit stark ein, hierdurch sind weniger Möglichkeiten vorhanden die betrachtet werden und durch die Zeitersparnis ist eine tiefere Suche möglich. Jedoch kann dieses Verfahren auch das Problem mit sich bringen, dass der Suchbereich zu stark eingengt und die beste Möglichkeit gar nicht mehr betrachtet wird. Zu verschiedenen Zeitpunkten im Spiel gibt es auch unterschiedlich viele Möglichkeiten zu ziehen, was auch kartenspezifisch ist. Dies sollte ähnlich einer Gauß-Kurve verlaufen. Am Anfang gibt es wenig Möglichkeiten, in der Mitte des Spiels am meisten und am Ende des Spiels kann man auf fast kein Feld mehr ziehen, wodurch es wieder wenige Möglichkeiten gibt.

Hier haben wir uns entschieden, diese Kurve durch einen Sinus-Verlauf zu nähern. Der Spielverlauf wird auf den Bereich von Null bis Pi betrachtet. Da wir das Fenster nicht auf 0 setzen wollen, haben wir hier eine leichte Verschiebung vorgenommen zu $\sin(x) * 0.2$.

Das x repräsentiert hierbei der Fortschritt im Spiel, also die aktuelle Spielrunde. Der aus dem Sinus resultierende Wert wird noch mit einem Faktor multipliziert, den wir aus der Kartengröße berechnen. Die Vor- und Nachteile der Aspiration-Windows werden in Abb. 15 verdeutlicht. Interessant ist besonders die maximal erreichte Tiefe, die wir leider nicht analysieren konnten, da ohne die Aspiration-Windows meist der von uns vorgegebene Maximalwert für die Tiefe

(42) erreicht wurde was eigentlich nicht der Fall sein sollte, siehe Abb. 15. Nur auf der Karte "thermonuk.map" lässt sich erkennen, dass sowohl die durchschnittliche Tiefe als auch die maximal erreichte Tiefe verbessert wurden. Zudem ist von Interesse, wie viel besser wir mit den Aspiration-Windows spielen, da dies das Kriterium ist, ob wir die Aspiration-Windows weiterhin nutzen oder deaktivieren.

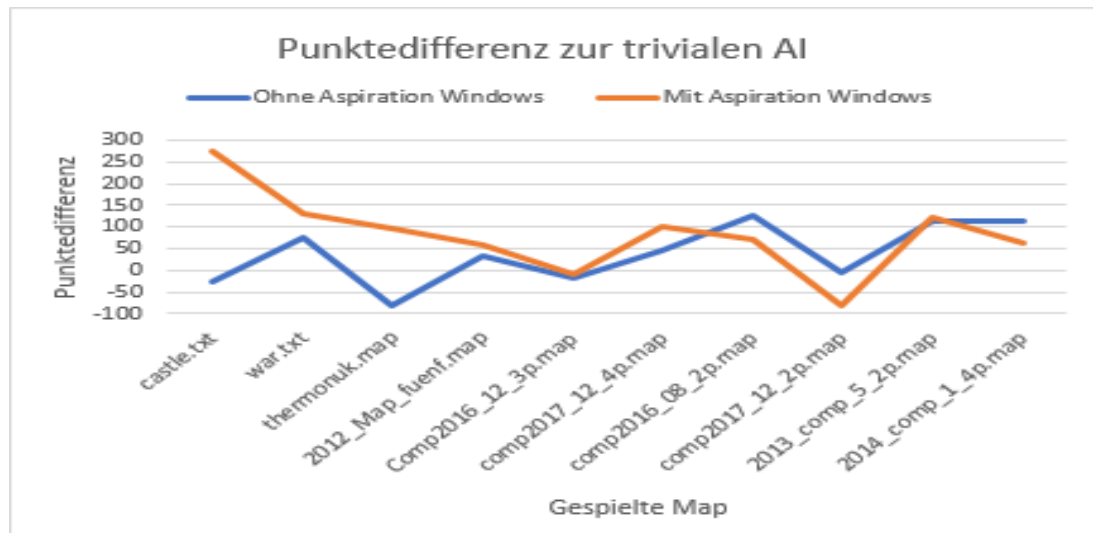


Abbildung 7: Vergleich der Punktedifferenz zur trivialen AI mit und ohne Aspiration-Windows

In Abb. 7 lässt sich erkennen, dass unsere KI auf fünf Karten besser spielt als zuvor, auf zwei Karten vergleichbar und auf 3 Karten schlechter als zuvor, da durch die Aspiration-Windows das Betrachtungsfeld für Züge von der KI stark eingegrenzt wird und sowohl gute als auch schlechte Züge nicht mehr betrachtet werden. Somit kommen wir im Idealfall näher an das wirkliche Spielgeschehen im Spiel von mehr als 2 Spielern, verlieren aber sehr gute Zugoptionen und lassen Möglichkeiten außer Acht, dass uns der Gegner bei bestimmten Zügen stärker einschränken kann als wir dies gern hätten. Da wir jedoch über mehrere Karten eine positive Bilanz haben, entscheiden wir uns dafür, die Aspiration-Windows weiterhin zu benutzen.

4.5 Parametrierung

Nun ist die KI sowohl im Zeitverhalten als auch von der Heuristik bereits sehr leistungsfähig. Jedoch kann man immer noch weiter optimieren. Viele Zusammenspiele von Parametern kann der Mensch selbst nicht mehr begreifen. Auch wenn die Logik schlüssig scheint, führt die KI noch fehlerhafte Züge aus. Dies kann eine Folge davon sein, dass die Parameter nicht richtig durchdacht wurden, oder dass die Parameter bereits im Bereich des "Overfitting" liegen und Einzelfälle zu viel Gewicht erhalten. Um dies herauszufinden, haben wir uns dazu entschieden zwölf Parameter im Bereich um die von uns gedachten Werte auszutesten.

Schritt 1) Auswahl von schwer schätzbaren Parametern

Da nicht unbegrenzt Rechenleistung bzw. genug Zeit existiert um alle möglichen Auswirkungen von Parametern auf die KI zu testen, haben wir uns für zwölf Parameter entschieden, deren Auswirkungen sich am schwierigsten abschätzen lassen und von denen wir uns die größten Verbesserungen erhoffen.

Diese Parameter sind im Folgenden aufgelistet.

- AspirationWindows-Nutzung ja oder nein
- Minimalbreite des AspirationWindows
- Faktor für die Breite des AspirationWindows nach der gedachten Berechnung
- Zeitlimit des Zuges, ab wann eine nächste Tiefe des IterativeDeepening gerade noch begonnen wird
- Sprungweite des Iterative Deepening
- Entfernung zwischen den Spielern um zu entscheiden, welcher Spieler bevorzugt angegriffen wird
- Wert einer Bombe bei betreten eines Bonussteins im Multiplayer-Modus (drei oder mehr Spieler)
- Wert eines Überschreibsteins bei betreten eines Bonussteins im Multiplayer-Modus
- Wert eines Choice-Feldes im Multiplayer-Modus (drei oder mehr Spieler)
- Wert des Heuristic-Values in Form einer fallenden Funktion im Multiplayer-Modus
- Wert der Anzahl eingenommener Steine in Form einer steigenden Funktion im Multiplayer-Modus
- Break-Even-Point im Spielverlauf ab wann die Anzahl der Steine im Spiel wichtiger ist, als die Position auf dem Spielfeld

Schritt 2) Erste Auswahl geeigneter Parameterwerte

Hierzu geben wir in einer Konfigurationsdatei eines externen Programms an, welche Parameter wir betrachten wollen, mit dem von uns gedachten Default-Wert sowie dem Intervall und die

Anzahl angegebener Schritte unter und über dem Default-Wert. Hierbei entstehen bis zu acht Einstellungsmöglichkeiten für jeden Parameter, wobei der Default-Wert jeweils erhalten bleibt. Für jeden Parameter haben wir nun acht Dateien, die sich nur in dem zu betrachtenden Parameter unterscheiden. Aus diesen 8 Einstellungsmöglichkeiten werden mithilfe eines Skripts, dass die Variablen in den Java-Code kopiert und kompiliert, 8acht lauffähige KIs erzeugt. Diese spielen auf drei möglichst unterschiedlichen Maps gegeneinander in zwei bis acht Spielerkarten. Die zwei besten KIs gewinnen und werden in den dritten Schritt übernommen.

Schritt 3) Auswahl von Parameterkombinationen

Aus dem zweiten Schritt haben wir nun zwölf Parameter mit je zwei möglichen Werten erhalten. Nun widmen wir uns der Tatsache, dass jeder Parameterkombination die KI verschlechtern oder auch verbessern kann. Dies ist jedoch kartenspezifisch. Uns interessieren aber keine Einzelfälle, sondern unser Ziel ist es auf möglichst vielen verschiedenen Maps gut zu spielen. Daher bilden wir alle Parameterkombinationen. Dies sind zwei hoch zwölf Stück, also 4098 Kombinationen. Diese spielen in zufällig ausgewählten achter Gruppen auf fünf verschiedenen Maps, die auch in Kurs-Matches benutzt wurden. Der jeweiliger Gewinner kommt eine Runde weiter, der Rest der Konfigurationen wird gelöscht. Somit haben wir bereits hier ca. 500 Spiele auf Zeit. Aus dem Kurs wissen wir, dass 70 Matches mit Zeitlimit von zwei Sekunden ungefähr 20 Minuten dauern. Das heißt dieser Schritt dauert überschlagen und mit etwas Puffer grob 150 Minuten, also 2,5 Stunden. Nun haben erhalten wir 512 restliche Kombinationen, somit 64 weitere Spiele. Diese sollten in weniger als einer Stunde machbar sein. Dies machen wir nun noch einmal, somit erhalten wir 8 übrig bleibende KIs. Diese werden nun in möglichst vielen Maps gegeneinander geprüft, wobei hier die "Triviale AI" ebenso aufgenommen wird. Nachdem alle möglichen Kombinationen gespielt haben, erhalten wir einen Sieger, dessen Werte wir in unsere KI aufnehmen werden.

5 Bombenphase

Sobald das Spiel in die 2. Phase (Bombenphase) wechselt, sind bereits alle Steine aller Spieler gesetzt. Ein weiteres Setzen ist nicht mehr möglich. Hier geht es nun darum, mit der spielfeld-spezifischen Anzahl an Bomben, sowie eventuell durch Bonusfelder erhaltener Zusatzbomben die eigene Vormacht auf dem Spielfeld zu stärken, indem man gegnerische Spielsteine weg-bombt. Diese Spielfelder werden aus dem Spielfeld gelöscht und werden zu Löchern im Feld. Wie viele Spielfelder aus dem Spielfeld entfernt werden hängt von der Bombenstärke ab, die zu Spielbeginn kartenspezifisch vorgegeben ist. Die Zahl gibt dabei an, wie weit ein Feld vom Detonationsort der Bombe entfernt sein darf, um gelöscht zu werden. Spielfeldlöcher bilden hierbei eine Barriere.

Beispiel:

```
1 - 0 1 2
- - 2 2 2
0 1 0 2 2
0 0 0 2 2
0 0 0 2 2
```

mit einer Transition von unten rechts nach oben rechts sowie der Bombenstärke 3. Wir werfen nun auf die Position (y: 0, x: 2, nullindiziert). Das Feld sieht nun wie folgt aus:

```
1 - - - -
- - - - -
- - - - -
- - - - -
0 0 0 2 -
```

Die 1 auf der Position (0,0) wurde durch die umgehenden Löcher geschützt.

Unsere Strategie ist die folgende: das Feld ist ab dem Erreichen der Bombenphase fix, nur dass manche Felder zu Löchern werden. Eine Vorhersage zu treffen ist nur schwer möglich, daher lassen wir die Zukunftsvorhersage (Paranoid-Algorithmus) weg und suchen uns nur den Zug heraus, der unsere Lage auf der Map verbessert, indem wir möglichst viele gegnerische Steine

von der Map entfernen. Hierbei machen wir zu Beginn jeder Runde eine kurze Analyse, welcher Spieler wie gut aufgestellt ist. Hierbei merken wir uns sowohl den nächstbesseren als auch den nächstschlechteren Spieler und die jeweilige Information, wie groß der Abstand des Spielers zu uns ist (Einzige Grundlage: Spielsteine zählen). Danach suchen wir uns aus, auf welchen von beiden Spielern wir unseren Fokus legen.

Beispiel:

Spieler 1 hat 670 Steine.

Wir haben 260 Steine.

Spieler 3 hat 250 Steine.

Spieler 4 hat 50 Steine.

Jeder Spieler hat eine Bombe mit der Stärke 1.

Der Spieler vor uns ist Spieler 1, der Spieler nach uns ist Spieler 3, den Rest beachten wir nicht weiter. Wenn alle Spieler den Spieler 1 attackieren wird er nicht genug Steine verlieren, damit wir ihn noch einholen könnten, Spieler 3 ist jedoch nur knapp hinter uns. Wenn wir nun sehr stark getroffen werden, verlieren wir einen Rang und bekommen somit deutlich weniger Punkte in dieser Runde. Somit setzen wir unseren Fokus auf Spieler 3 um diesen so gut wie möglich zu schwächen, damit uns dieser Spieler nicht mehr so einfach einholen kann.

Der Algorithmus lautet nun wie folgt:

Versuche jeden Stein der auf dem Spielfeld existiert (nicht '-') zu treffen. Merken, welcher Spieler wie viele Steine verloren hat sowie die Summe aller Steine die die Gegner verloren haben. Haben bei der nächsten Betrachtung die Gegner mehr oder gleich viele Steine verloren als zuvor, schaue dir an ob die Differenz "Gegnerische Steine - Unsere Steine" größer oder gleich ist als zuvor. Wenn ja: Hat der favorisierte Gegner mehr Steine verloren als zuvor? Ja: merke dir diese Position. Wenn eine Instanz falsch zurückgibt, schaue dir die nächste Position an.

Um die Auswirkung einer Bombe an einer Position korrekt zu ermitteln, nutzen wir eine rekursive Funktion. Besitzt die Bombe eine Stärke von 3 gehen wir in jede mögliche Richtung ein Feld weiter und setzen dort jeweils eine Bombe der Stärke 2. Dies geht solange weiter, bis wir eine Bombenstärke von 0 erreichen, dies bedeutet, dass dieses Feld durch ein Loch ersetzt wird. Leider könnte dies Auswirkungen auf andere Richtungen haben, sodass manche Felder nicht weggesprengt werden, da ein '-' eine Barriere bildet. Somit mussten wir ein temporäres Zei-

chen auf der map 't' einführen um dieses Problem zu umgehen. Würden wir dies nicht machen sondern einfach die ursprünglichen Zeichen lassen, hätten wir das Problem dass wir manche Felder mehrfach zählen, was das Ergebnis verfälscht, wie viele eigene und gegnerische Steine wir wegsprengen. Die 't' werden nach dem Setzen der endgültigen Bombe (Information durch Server) einfach durch '-' ersetzt.

Für die rekursive Variante spricht, dass wir eine kleine Funktion haben, die immer wieder mit ähnlichen Parametern aufgerufen wird (andere Bombenstärke, anderes x und y). Somit sollte es dem Compiler möglich sein, die Funktion im Cache zu lassen und es muss bei jedem Schritt nur wenig abgeändert werden. Wichtig ist für die Funktion nur: x-Koordinate der Detonation, Y-Koordinate der Detonation, Bombenstärke lesend, char am Spielfeld(y,x) lesend und schreibend, Array mit Länge [Spielerzahl + 1] um abzuspeichern welcher Spieler wie viele Steine verloren hat, die Position 0 gibt die Gesamtanzahl der Steine an die alle Gegner verloren haben. Dadurch dass sich bereits große Teile der Funktion im Cache befinden sollten, ist die Abarbeitungsgeschwindigkeit sehr hoch.

6 Wettbewerbs-Spielfelder

Unsere KI ist besonders auf den Maps stark, auf denen viele Spezialsteine existieren. Bonusfelder, Choicefelder, Inversionsfelder und Überschreibsteine nutzen wir in der Regel besonders effizient. Schwierig wird es für uns, wenn wir Choice oder Inversionssteine erkennen, diese im Spiel aber nicht mehr erreichen, da das Spiel zuvor abgebrochen wurde. Genauso hat unsere KI ein Problem mit der richtigen Positionierung, wenn es sehr viele Transitionen und nur wenige echte Ecken existieren, da wir noch nicht prüfen, wie sicher gesetzte Steine langfristig sind, daher werden viele von uns gesetzte Steine schnell wieder eingenommen. Daher planen wir Maps, die viele Sondersteine, wenig Transitionen und viele echte Ecken besitzen und auf denen uns viele angesammelte Überschreibsteine einen Vorteil bringen.

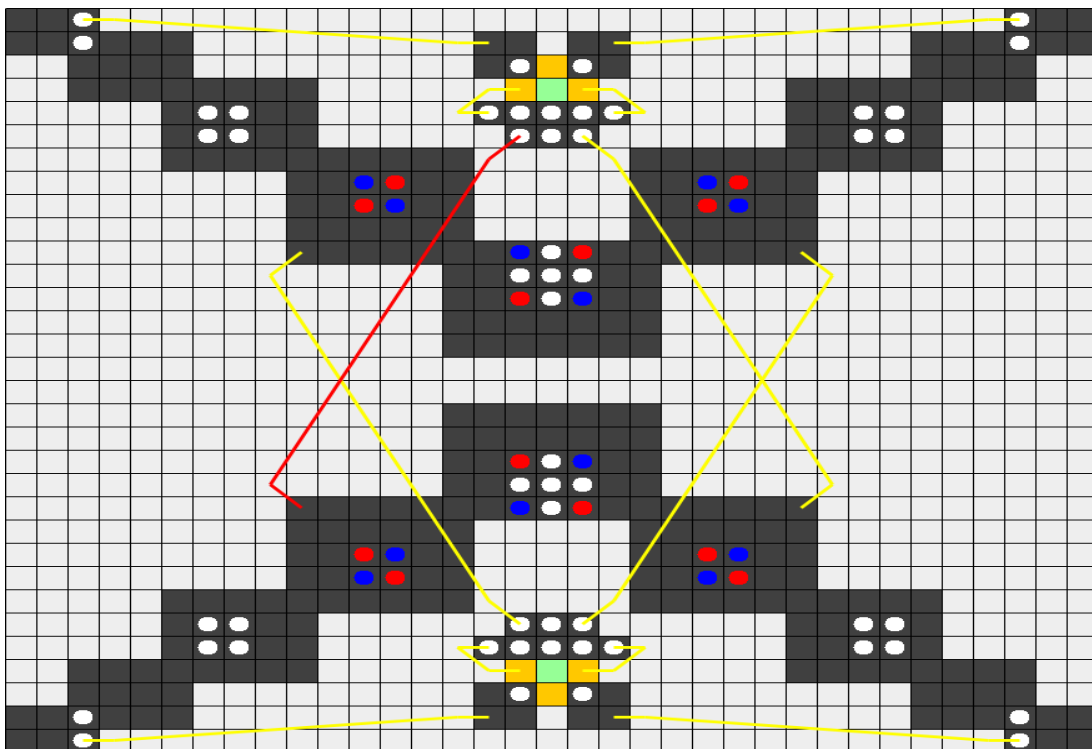


Abbildung 8: 2-Spieler-Karte

Auf der von uns gestalteten 2-Spieler-Karte, siehe Abb. 8 haben wir ein kleines Spielfeld designed, auf der es darauf ankommt, über die Flanken so schnell wie möglich zu dem Choice-Stein und zu den Bonussteinen zu kommen und diese taktisch gut einzusetzen. Der Spieler, der dies am schnellsten erkennt und mit der Situation am besten umgehen kann, ist klar im Vorteil. Da wir mit Sondersteinen gut umgehen können und trotzdem noch eine akzeptable Suchtiefe besitzen, sehen wir hier einen Vorteil für uns. Auf der Karte gibt es zu Beginn keine Überschreibsteine und keine Bomben. Zudem haben Bomben auf der relativ kleinen Map die Stärke 0, so dass es mehr auf die Taktik ankommt.

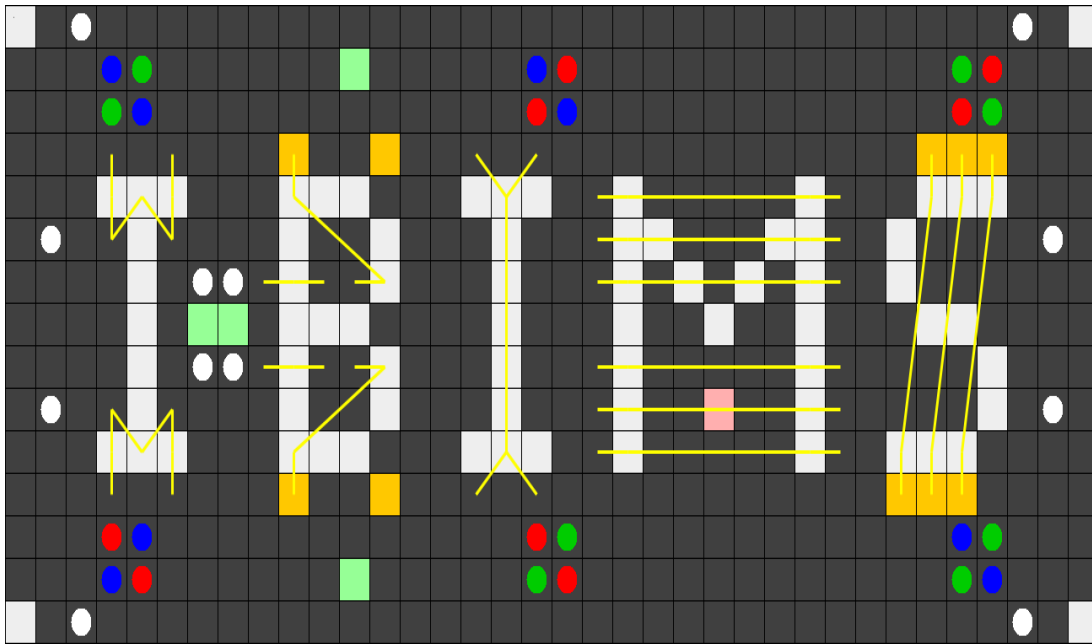


Abbildung 9: 3-Spieler-Karte

Auf der 3-Spieler-Karte, siehe Abb. 9 gibt es für jeden Spieler zu Beginn 3 Überschreibsteine, aber auch 2 Bomben mit Stärke 2. Die Bomben auf dieser Karte können viel Schaden anrichten. Jedoch können auch durch Überschreibsteine große Flächen der Karte kontrolliert werden, was dazu führt dass man sich genau entscheiden muss, für welche Option sich man bei den leicht zu erreichenden Bonusfeldern entscheidet. Auch gibt es eine Inversion und 4 Choice Felder, was den Spielausgang noch einmal kippen kann.

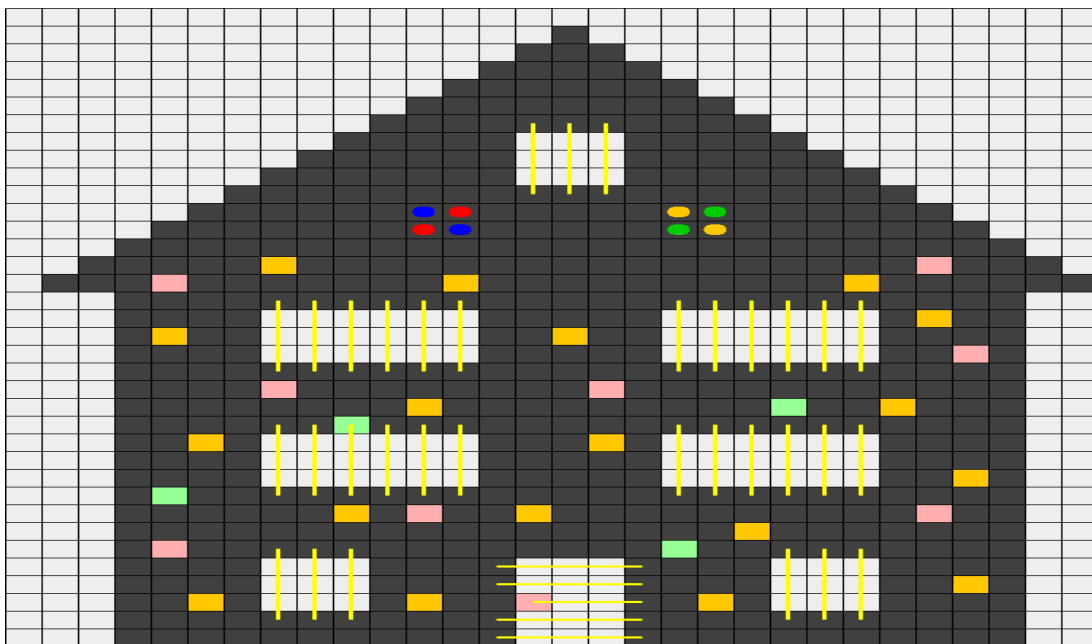


Abbildung 10: 4-Spieler-Karte

Auf der 4-Spieler-Karte, siehe Abb. 10 hingegen, spielen wir vor allem den Umgang mit vielen Spezialsteinen aus. Hier gilt es, auch über Transitionen hinweg, die Spezialsteine und deren Anzahl frühzeitig zu erkennen und für sich den besten Nutzen daraus zu ziehen sowie auch andere Spieler an deren Erreichen zu hindern. Der Umgang mit Spezialsteinen und eine hohe Suchtiefe sind hier durchaus ein großer Vorteil. Zudem ist es nützlich, sich Gedanken um die echten Ecken zu machen, auch in Kombination mit den Spezialsteinen. Auf dieser Karte hat jeder Spieler zu Beginn zwei Überschreibsteine und zwei Bomben der Stärke 2.

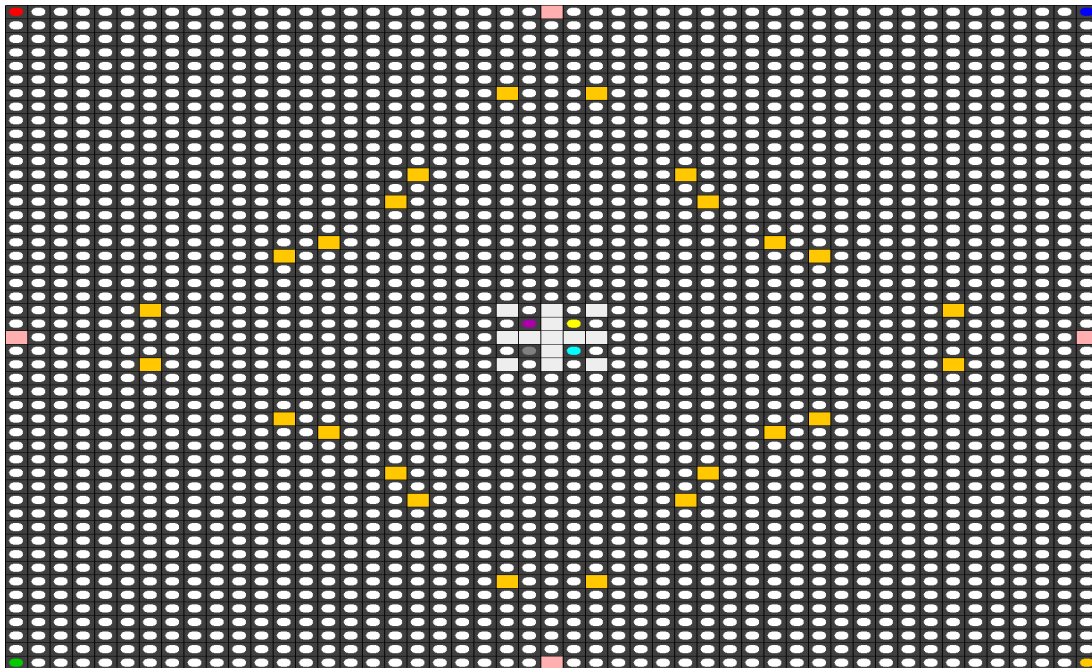


Abbildung 11: 8-Spieler-Karte

Auf der 8-Spieler-Karte, siehe Abb. 11, kommt es rein auf den Umgang mit Überschreibsteinen an. Es gibt auf der Karte zu Beginn keine Bomben, falls man bei einem Bonusstein eine Bombe wählt, hat diese eine Stärke von 1, was auf einer so großen Karte kaum Wirkung zeigt. Zudem besitzt jeder Spieler um sich einen Start verschaffen zu können 5 Überschreibsteine. Ziel ist es, sich diese Überschreibsteine gut einzuteilen und sich trotzdem gut auf der Karte zu positionieren um die Bonussteine sammeln zu können. Dass durch die Überschreibsteine viele eigene Steine verloren gehen können, sollte auf dieser Map klar sein, daher ist es wichtig, ein Muster zu bilden in dem viele Steine erhalten bleiben, auch dann wenn ein beliebiger Stein überschrieben wird. Hierfür sind viele Disziplinen notwendig: Umgang mit Überschreibsteinen, rechnen auf Tiefe um viele Optionen vorhersehen zu können, und ein Spiel auf Sicherheit der eigenen Steine. Vor allem, da jeder Spieler mit nur einem Stein in einem echten Eck beginnt (dieser Stein ist jedoch die erste Zeit sicher, dass man nicht unverschuldet auf 0 Steine kommt).

7 Logging

Um Fehler besser verfolgen zu können, haben wir uns einen aktivierbaren Logger-Manager geschrieben. Dieser untergliedert sich in mehrere Bestandteile:

Erstens gibt es hier eine config-Datei die im config-Ordner vorzufinden ist. Hier wird der Pfad zu dem Ordner angegeben, an dem die Logs gespeichert werden sollen.

Zweitens haben wir verschiedenen Log-Stufen eingebaut. Der Logging-Modus wird mit dem Startparameter `-log` aktiviert.

Die erste Stufe des Logging ist mit dem Startparameter `+l0` erreichbar. Hier wird nur das Endergebnis in einer csv-Datei gespeichert. Dies funktioniert Spielübergreifend.

Die zweite Stufe ist mit dem Startparameter `+l1` aktivierbar. Jeder abgesendete eigene Zug dieses Spiels wird detailliert abgespeichert. Das Log enthält nach Datum und Uhrzeit ein `l1` im Dateinamen.

Die dritte Stufe ist mit dem Startparameter `+l2` aktivierbar. Jeder mögliche Zug im Paranoid-Algorithmus wird detailliert abgespeichert. Das Log enthält nach Datum und Uhrzeit ein `l2` im Dateinamen. Dieser Modus kostet bereits sehr viel Zeit und sollte nur verwendet werden, wenn man ein Problem auf andere Art und Weise nicht mehr nachvollziehen kann, wie z.B. die Verwendung der Überschreibsteine zu einem bestimmten Zeitpunkt.

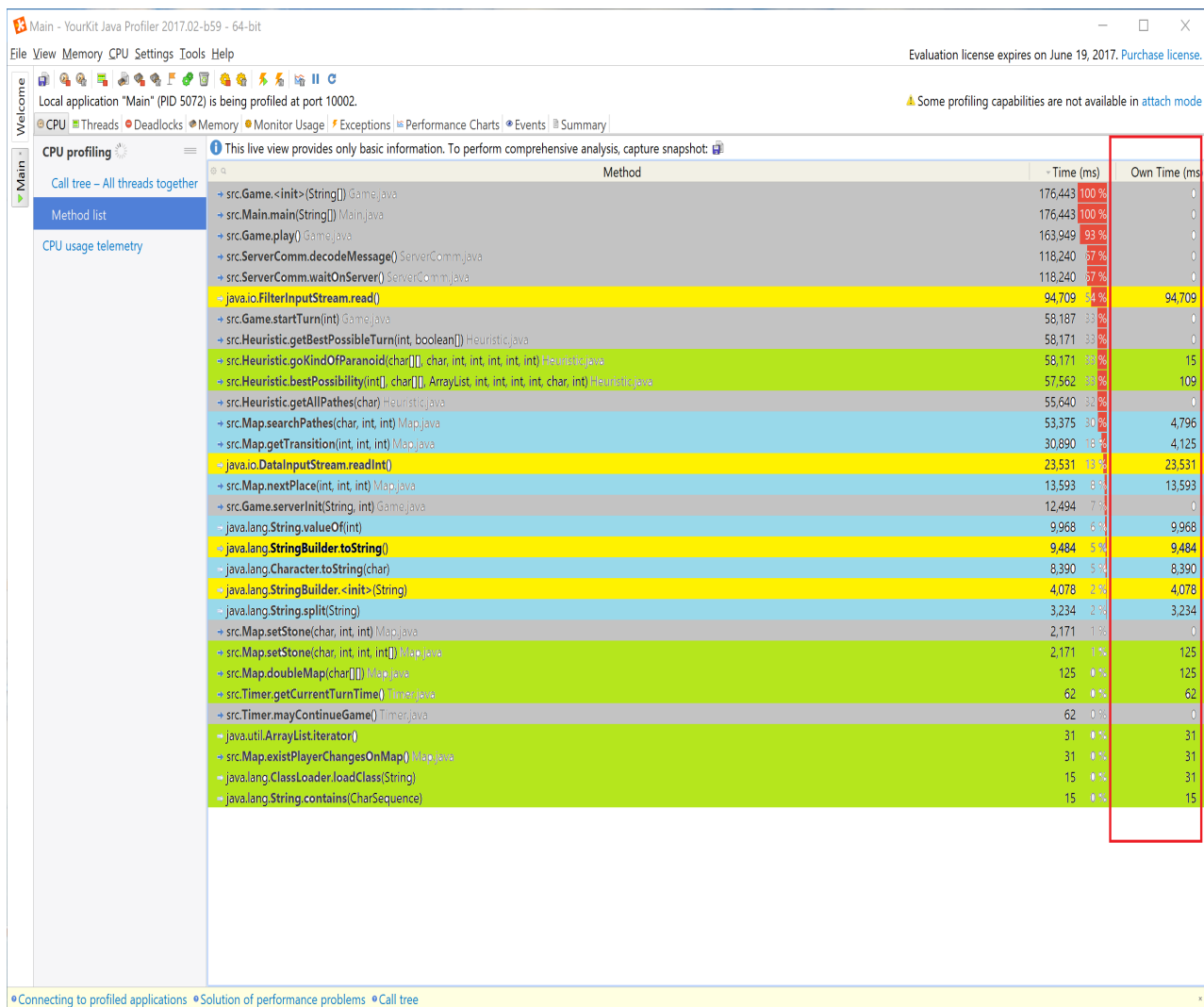
8 GUI

Um Spielzüge an bestimmten Positionen simulieren zu können sowie unser Einfärbe-Verhalten zu beobachten, haben wir zu Beginn des Projektes eine kleine GUI programmiert, siehe Abb. 16, die für jedes Feld auf der Map durch einen Button repräsentiert wird, um dort einen Stein setzen zu können. Die Buttons sind der Serverausgabe angepasst. Steine der Spieler Eins bis Acht bestehen aus einem weißen Feld mit einem Punkt der entsprechenden Farbe. Ein leeres Feld ist weiß, ein Loch schwarz. Die eigenen Spielsteine erhalten einen orangen Hintergrund. Sonderfelder sind ebenso gekennzeichnet. Leider sind Transitionen zum aktuellen Zeitpunkt noch nicht eingezeichnet. Auch die Legende, welcher Spieler welche Farbe besitzt ist noch nicht vorhanden. Jedoch lassen sich mit der GUI in Spielen ohne Zeit spezielle Spielzüge ausprobieren um das korrekte Einfärbeverhalten unserer KI zu testen. Die GUI ist über zwei Schnittstellen an das eigentliche Programm gekoppelt: die Serverkommunikation um einen Zug abschicken zu können und die Map um das Spielfeld aufzubauen. Die GUI hat zwei Modi. Zum einen der automatische Modus um dem Spiel folgen zu können, indem man zuschaut, zum anderen der manuelle Modus um selbst Steine setzen zu können. Dies hat jedoch die Schwachstelle, dass im manuellen Modus das Berechnen des besten Zuges sowie das Absenden dieses Zuges an den Server unterbrochen werden muss. Daher macht sich der Wechsel des Modus erst beim nächsten Zug den man ziehen könnte bemerkbar. Zudem kostet das Ausführen der GUI erhöhte Ressourcen, vor allem Berechnungszeit, weshalb die GUI in Spielen auf Zeit nicht aktivierbar ist.

Die GUI wird mit dem Startparameter `-gui` mitgestartet und ist ansonsten deaktiviert. Mit dem Startparameter `-manual` startet die GUI zudem in dem Modus, dass man als Spieler von Anfang an selbst spielen kann, der Default-Modus ist der automatische Spielablauf, bei dem man der KI nur beim Spielen zuschaut.

9 Profiling

Nachdem die KI ohne Disqualifikationen spielt, widmen wir uns der Optimierung weniger Zeit für Zug zu verbrauchen und statt dessen tiefer rechnen zu können. Daher haben wir eine 15-tägige Testversion des YourKit-Profilers heruntergeladen, der unsere KI während eines ganzen Spiels analysiert. Hierbei wird uns das folgende Ergebnis aus Abbildung 12 angezeigt:



Method	Time (ms)	Own Time (ms)
src.Game.<init>(String[]) Game.java	176,443	100 %
src.Main.main(String[]) Main.java	176,443	100 %
src.Game.play() Game.java	163,949	93 %
src.ServerComm.decodeMessage() ServerComm.java	118,240	57 %
src.ServerComm.waitOnServer() ServerComm.java	118,240	57 %
java.io.FilterInputStream.read()	94,709	4 %
src.Game.startTurn(int) Game.java	58,187	33 %
src.Heuristic.getBestPossibleTurn(int, boolean[]) Heuristic.java	58,171	33 %
src.Heuristic.goKindOfParanoid(char[], char, int, int, int, int, int) Heuristic.java	58,171	33 %
src.Heuristic.bestPossibility(int[], char[], ArrayList, int, int, int, char, int) Heuristic.java	57,562	29 %
src.Heuristic.getAllPathes(char) Heuristic.java	55,640	32 %
src.Map.searchPathes(char, int, int) Map.java	53,375	30 %
src.Map.getTransition(int, int, int) Map.java	30,890	18 %
java.io.DataInputStream.readInt()	23,531	13 %
src.Map.nextPlace(int, int, int) Map.java	13,593	8 %
src.Game.serverInit(String, int) Game.java	12,494	7 %
java.lang.String.valueOf(int)	9,968	6 %
java.lang.StringBuilder.toString()	9,484	5 %
java.lang.Character.toString(char)	8,390	5 %
java.lang.StringBuilder.<init>(String)	4,078	2 %
java.lang.String.split(String)	3,234	2 %
src.Map.setStone(char, int, int, int) Map.java	2,171	1 %
src.Map.setStone(char, int, int, int) Map.java	2,171	1 %
src.Map.doubleMap(char[]) Map.java	125	0 %
src.Timer.getCurrentTurnTime() Timer.java	62	0 %
src.Timer.mayContinueGame() Timer.java	62	0 %
java.util.ArrayList.iterator()	31	0 %
src.Map.existsPlayerChangesOnMap() Map.java	31	0 %
java.lang.ClassLoader.loadClass(String)	15	0 %
java.lang.String.contains(CharSequence)	15	0 %

Abbildung 12: Profiling unserer KI über ein komplettes Spiel, überarbeitet

Dieses Ergebnis wurde farblich nachbearbeitet. Zu bearbeitende Funktionen haben eine hohe benötigte Zeit in der Spalte "Own Time". Gelbe Zeilen kommen von Standard-Implementierungen in Java. Eine Verbesserung wäre nur möglich, indem man diese Funktionen umgeht. Der "Input-Stream" wird von der Serververbindung unseres Clients benötigt. Die String-Methoden kosten viel Zeit, nach einem Test verbrauchen wir jedoch deutlich weniger Zeit, als wenn wir die Strings,

die nur für die Transitionen benutzt werden, durch Arrays oder Listen ersetzen würden. Die grauen Methoden sind bereits sehr effektiv, sie selbst verbrauchen kaum eigene Zeit. Ebenso die grünen Methoden, sie benötigen so wenig eigene Zeit, dass sie nicht in unserem Fokus liegen. Anders sieht es jedoch mit den blau markierten Methoden aus. Die String-Methoden haben wir bereits diskutiert, diese können wir nicht ändern, sie sind eine Standard-Java-Implementierung, bereits sehr effizient und von uns benötigt. Die Funktionen "nextPlace "und "getTransition "sind sehr elementar, werden jedoch sehr oft benötigt. Die Methode "searchPath "verbraucht unserer Meinung nach zu viel Zeit. Das Problem waren Listen die innerhalb von Schleifen deklariert und nicht nur initialisiert wurden. Die Effizienz konnte erheblich gesteigert werden. Dies ist in Zahlen jedoch nicht mehr nachweisbar, da unsere Lizenz des Profilers abgelaufen ist.

10 Fazit

Das ganze Team ist der Meinung, dass uns das Fach sehr gut gefallen hat und wir viel neues und auch nützliches gelernt haben. GIT und ANT waren uns vor dem Projekt gänzlich unbekannt und wir glauben, dass es nützlich ist diese Programme nun zumindest ein wenig beherrschen zu können. Weiterhin hat uns das Projekt die grundsätzlichen Überlegungen beim KI programmieren beigebracht welche man ja auch auf andere KIs in anderen Bereichen anwenden kann. Für jeden aus dem Team war dies das erste Gruppen-Programmier-Projekt. Wir haben interne Unstimmigkeiten zur Lösung mancher Probleme überwinden müssen und jeder von uns hat etwas an Kompromissbereitschaft eingebracht. Diese Diskussionen waren auch sehr förderlich, da wir gemeinsam viele Fehler aber auch neue Lösungen gefunden haben die für einen alleine sehr schwierig zu erkennen gewesen wären.

Wir haben auch erfahren dass es teils schwierig sein kann, wenn mehrere Personen am selben Code arbeiten. Man verliert teils die Übersicht oder man versteht den Code des Teamkollegen nicht. Durch Kommentare im Code, aber vor allem durch ausführliche Kommunikation im Team haben wir auch dieses Problem überwunden.

Der Aufwand den wir für das Projekt betreiben mussten orientierte sich im Groben an der Kurve welche uns von Prof. Dr. Carsten Kern am beginn der Projektes vorgestellt wurde. Jedoch empfanden wir den Aufwand nicht als zu groß, da uns das Projekt sehr viel Spaß gemacht hat.

Anhang

A Abbildungsverzeichnis

Daten:														
	Kontrahent:	triviale AI / eigene KI												
	Server-Zeitlimit:		1 Sekunde / Zug						1 Sekunde / Zug					
Map			Ohne Iterative Deepening						Mit Iterative Deepening					
			Durchschnittliche Zustände / Runde	Durchschnittliche Zeit [ms] / Runde	Durchschnittliche Zeit [ms] / Zustand	Durchschnittliche Tiefe	Maximale Tiefe	Differenz Wir, Gegner; >0 gewonnen, <0 verloren	Durchschnittliche Zustände / Runde	Durchschnittliche Zeit [ms] / Runde	Durchschnittliche Zeit [ms] / Zustand	Durchschnittliche Tiefe	Maximale Tiefe	Differenz Wir, Gegner; >0 gewonnen, <0 verloren
castle.txt			23967	2361	0,098	6,9	42	-5	71180	3009	0,042	6,5	13	276
war.txt			62078	2516	0,04	6,2	42	133	116268	2849	0,024	5,6	27	129
thermonuk.map			32695	2419	0,074	5,9	42	54	137940	2589	0,018	5,6	13	95
2012_Map_fuenf.map			112814	2225	0,019	5,6	42	52	133825	2826	0,021	5,2	9	59
Comp2016_12_3p.map			105891	2287	0,021	10,2	42	76	38208	3589	0,093	9,2	15	-10
comp2017_12_4p.map			144762	2180	0,017	5,7	42	45	268755	2685	0,009	5,3	25	101
comp2016_08_2p.map			97635	2469	0,032	3,6	42	15	145033	2439	0,017	3,2	9	73
comp2017_12_2p.map			107482	2647	0,041	7,8	42	-102	95454	3270	0,034	7,5	25	-82
2013_comp_5_2p.map			186482	2148	0,022	4,1	42	107	275759	2445	0,008	3,7	7	120
2014_comp_1_4p.map			87479	3516	0,055	6,7	42	54	152025	5208	0,034	6,2	15	61

Abbildung 13: Vergleich ohne und mit Iterative Deepening

Daten:		Kontrahent:	triviale AI / eigene KI													
		Server-Zeitlimit:	1 Sekunde / Zug							1 Sekunde / Zug						
Map			Ohne Zugsortierung							Mit Zugsortierung						
			Durchschnittliche Zustände / Runde	Durchschnittliche Zeit / Runde	Durchschnittliche Zeit / Zustand	Durchschnittliche Tiefe	Maximale Tiefe	Differenz Wirr, Gegner: >0 gewonnen, <0 verloren	Durchschnittliche Zustände / Runde	Durchschnittliche Zeit / Runde	Durchschnittliche Zeit / Zustand	Durchschnittliche Tiefe	Maximale Tiefe	Differenz Wirr, Gegner: >0 gewonnen, <0 verloren		
castle.txt			25850	3226	0,125	4,8	6	-8	71180	3009	0,042	6,5	13	276		
war.txt			26012	3076	0,118	5,3	8	2	116268	2849	0,024	5,6	27	129		
thermonuk.map			50190	2904	0,057	4,5	10	34	137940	2589	0,018	5,6	13	95		
2012_Map_fuenf.map			118960	3269	0,027	5,5	30	-2	133825	2826	0,021	5,2	9	59		
Comp2016_12_3p.map			50420	2929	0,058	5,8	12	-34	38208	3589	0,093	9,2	15	-10		
comp2017_12_4p.map			81547	2965	0,036	4,9	18	78	268755	2685	0,009	5,3	25	101		
comp2016_08_2p.map			89946	2804	0,031	4,6	20	131	145033	2439	0,017	3,2	9	73		
comp2017_12_2p.map			144775	3791	0,026	6,4	16	29	95454	3270	0,034	7,5	25	-82		
2013_comp_5_2p.map			124409	3058	0,024	4,2	14	11	275759	2445	0,008	3,7	7	120		
2014_comp_1_4p.map			19540	3064	0,156	7,5	18	67	152025	5208	0,034	6,25	15	61		

Abbildung 14: Vergleich mit und ohne Zugsortierung

Daten:														
	Kontrahent:	triviale AI / eigene KI												
	Server-Zeitlimit:		1 Sekunde / Zug						1 Sekunde / Zug					
Map			Ohne Aspiration Windows						Mit Aspiration Windows					
			Durchschnittliche Zustände / Runde	Durchschnittliche Zeit / Runde	Durchschnittliche Zeit / Zustand	Durchschnittliche Tiefe	Maximale Tiefe	Differenz Wir, Gegner; >0 gewonnen, <0 verloren	Durchschnittliche Zustände / Runde	Durchschnittliche Zeit / Runde	Durchschnittliche Zeit / Zustand	Durchschnittliche Tiefe	Maximale Tiefe	Differenz Wir, Gegner; >0 gewonnen, <0 verloren
castle.txt			76083	2373	0,031	9,1	42	-24	71180	3009	0,042	6,5	13	276
war.txt			123388	2515	0,02	6,7	42	76	116268	2849	0,024	5,6	27	129
thermonuk.map			132602	2415	0,018	3,7	8	-80	137940	2589	0,018	5,6	13	95
2012_Map_fuenf.map			266477	2378	0,009	6	42	34	133825	2826	0,021	5,2	9	59
Comp2016_12_3p.map			112036	2387	0,021	8,8	42	-16	38208	3589	0,093	9,2	15	-10
comp2017_12_4p.map			264374	2418	0,012	5,1	42	47	268755	2685	0,009	5,3	25	101
comp2016_08_2p.map			80351	2441	0,03	4,6	42	128	145033	2439	0,017	3,2	9	73
comp2017_12_2p.map			34923	1927	0,055	13,6	42	-5	95454	3270	0,034	7,5	25	-82
2013_comp_5_2p.map			147560	2443	0,016	4,55	42	113	275759	2445	0,008	3,7	7	120
2014_comp_1_4p.map			40851	2451	0,06	5,6	42	115	152025	5208	0,034	6,25	15	61

Abbildung 15: Vergleich mit und ohne Aspiration Windows

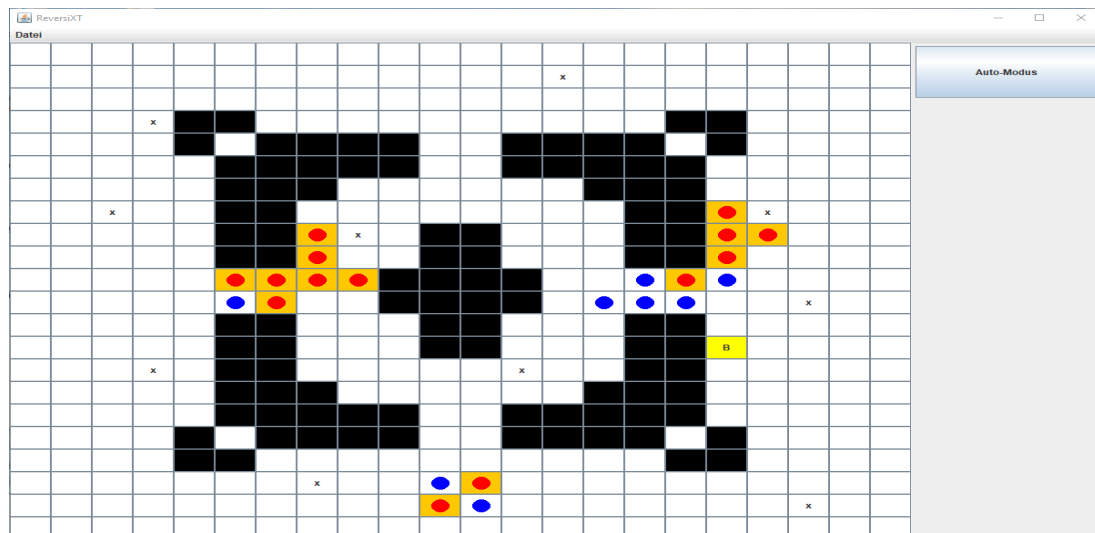


Abbildung 16: GUI zum beobachten oder auch zum mitspielen