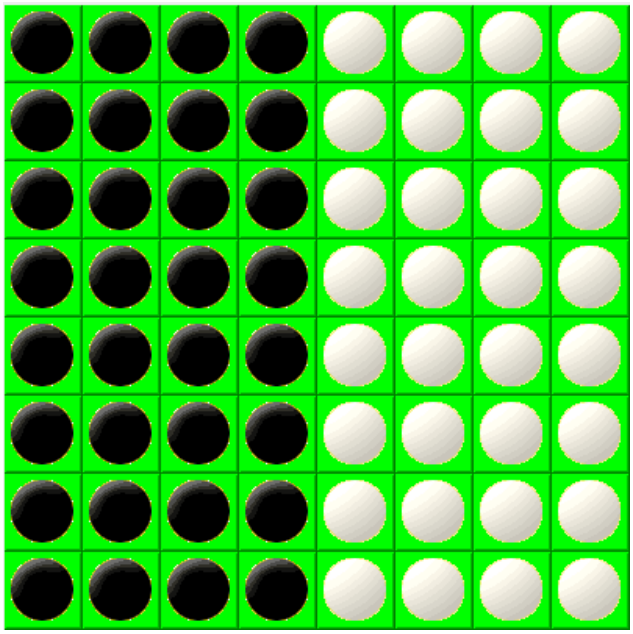
Zürcher Hochschule für angewandte Wissenschaften

Diplomstudium Informatik



Projektarbeit 4. Semester

Informatikprojekt

JReversi

Autoren	Oliver Aeschbacher René Kamer
Dozent	Jens-Christian Fischer
Projektstart	26. März 2012
Projektpräsentation	15. Juni 2012

Inhaltsverzeichnis

1. Abstract

Wir haben als Aufgabenstellung eine Reversi Implementation ausgewählt. Dieses Brettspiel soll einerseits gegeneinander spielbar sein sowie auch gegen einen Computergegner.
Es soll ein Schwierigkeitsgrad ausgewählt werden können, mit welcher danach der Computergegner spielt. Hierbei soll eine KI (Künstliche Intelligenz) zum Einsatz kommen, welche auf den gewählten Schwierigkeitsgrad zugeschnitten ist.
Ebenfalls soll eine Eröffnungsbibliothek implementiert werden, welche Eröffnungszüge korrekt anzeigt. Die vorliegenden Spielinformationen sollen übersichtlich dargestellt werden, d.h. gemachte Züge, wenn ein Spieler gepasst hat, Eröffnung sowie Gewinner bzw. Verlierer.
Es werden hohe Ansprüche an Aussehen und Animation gestellt.

2. Detailanalyse der Aufgabenstellung

Für die vorgegebene Aufgabenstellung werden verschiedenste Konzepte benötigt. Diese umfassen:

- GUI Implementierung des Spielfeldes sowie der Umgebenden Informationsanzeigen
- Korrekte Abbildung der Spielregeln des Spiels
- Algorithmen zur korrekten Traversierung des Spielfeldes
- Algorithmen zum Erkennen eines Passes (direkt ausgelöst, Spieler muss nicht einen Passknopf drücken)
- Abstrahieren der vorliegenden Spielinformationen in entsprechende Datenstrukturen
- Spielbaumtheorie
- Aufbauen einer Datenstruktur für einen Spielbaum
- Animation des Umdrehens der Spielsteine
- Bewertungsheuristik
- KI: Spielbaumtheorie im Zusammenhang mit dem Alpha Beta Algorithmus und deren Umsetzung

Die benötigten Konzepte werden einerseits selbst erarbeitet und andererseits aus der bekannten Spieltheorie abgeleitet.
Die selbst erarbeiteten Algorithmen müssen einen hohen Grad an Geschwindigkeit bieten, da man dem menschlichen Spieler keine sehr langen Wartezeiten zumuten will.
Die Datenrepräsentationen dürfen ebenfalls nur sehr wenig Platz beanspruchen da die Spielbaumtheorie eine speicherintensive Implementation darstellt bei einem komplexen Spiel wie Reversi.
Die Konzepte für Alpha Beta Algorithmen sind seit vielen Jahren bekannt und werden aus der Theorie direkt umgesetzt. Es wurde zuerst noch der MiniMax Algorithmus diskutiert, da jedoch die Alpha Beta Methode sehr viel Speicher spart, ist dies die erste Wahl geworden.

3. Motivation

Das Thema „Spieltheorie“ hat uns besonders fasziniert, weil wir uns beide leidenschaftliche an Logik- und Strategiespiele messen. Wir spielen beide Schach, kannten aber Reversi – auch wenn nur oberflächlich – bereits vor dem Projekt. Insbesondere haben wir uns für dieses Spiel entschieden, weil es einerseits ein sehr einfaches Spielprinzip besitzt, aber andererseits eine enorme strategische Leistung erfordert. Dies ist bei vielen anderen Spielen nicht der Fall.

4. Projektplanung

a. Prinzipielles Vorgehen

Als erstes stand die Planungsphase an. Möglichst alle zu implementierenden User Stories wurden erstellt und die dazugehörigen Tasks geschrieben. Da man am Anfang nicht alle Eventualitäten klären kann kamen im Verlauf des Projektes noch einige User Stories und Tasks hinzu.
Prinzipiell wurde als erstes das GUI realisiert, damit man visuell arbeiten kann und schon eine Diskussionsgrundlage hatte. Danach wurden sogleich die ersten Spiellogiken implementiert (Man könnte auch sagen, das Regelwerk wurde sukzessive umgesetzt).
Weitere Funktionen wurden hinzugefügt, so wie z.B. die Möglichkeiten, das Spiel entsprechend seinen Bedürfnissen einzustellen (Schwierigkeitsgrad, Menschlicher oder Computergegner, etc.).

Die Spiele-Anzeigen (letzter gemachter Zug, Passen, Spielgewinner) wurden noch vor schwierigeren Passagen (Spieltheorie, Animation) eingebaut, da dies noch relativ einfach zu machen sein ist.

Da wir auf dem Internet und in einschlägigen Foren praktisch keine animierten Versionen eines Reversis fanden, haben wir noch einen Animationsteil eingebaut (Zusätzliche User Story im Verlauf des Projekts), dies, um auch unseren eigenen Ansprüchen gerecht zu werden.

Um eine gute Basis nach der zweiten Iteration zu schaffen, haben wir ein grosszügiges Refactoring eingeplant. Dies, damit wir im nächsten Teil mit einer sauberen und aufgeräumten Umgebung arbeiten können.

Danach folgte die Umsetzung des eigentlichen Algorithmus, welcher den Kern des Projekts bildet. Dazu haben wir extra Zeit eingeplant, in welcher wir zuerst die gesamten Spieltheorien sowie die zugehörigen Algorithmen kennenlernen konnten. Danach konnten wir die Implementation der eben erwähnten Punkte in Angriff nehmen und das Projekt für erste externe Tests abschliessen.
Für diesen Teil haben wir uns ein wenig mehr Zeit herausgenommen, wissend dass wir hier noch auf Probleme und Verzögerungen stossen könnten.

Am Ende wurde das Programm noch relativ ausgiebig von externen Beta Testern auf Herz und Nieren geprüft. In der letzten Phase wurden die Aussagen der Tester nochmals auf das Programm übertragen bzw. angepasst und die letzten Dokumentationslücken geschlossen.

b. Iterationplan

Der Iterationsplan wurde online geführt, der Einfachheit halber ist dieser jetzt in seiner endgültigen Version in dieses Dokument übernommen worden. Die Tasks beziehen sich auf das externe Dokument „User Stories - Cards and Tasks.docx“ bzw. PDF.

i. Iteration 1

Iteration 1 Wochen 1-3					Mileston e 1
Layout: Spielfeld	Design + Funktion: Ziehender	Spielansicht: Darstellung der Steine	Spiellogik: Schlagen der Steine	Spiellogik: Setzen eines Steines	Spiellogik: Passen
Task1 (OA)	Task4 (RK)	Task18 (OA)	Task22 (RK)	Task30 (OA)	Task27 (OA) Light
Task2 (OA)	Task5 (OA)	Task19 (RK)	Task23 (RK)	Task31 (RK)	Task28 (OA)
Task3 (RK)	Task6 (RK)		Task24 (OA)		Task29 (RK)

Legende:

User Stories	Tasks	Milestones	In Progress	Abge-schlossen

ii. Iteration 2

Iteration 2 Wochen 4-6					Mileston e 2
Spiellogik: Wo dürfen Steine gesetzt werden	Startseite: Spieleinstellungen	Design + Funktion: Zug Angabe	Ansicht: Anz. der Steine, welche umgedreht werden	Design + Funktion: Spielkontrolle	Ansicht: Anzeige der Eröffnung
Task 20 (RK)	Task 16 (OA)	Task 7 (OA)	Task 25 (OA)	Task 10 (OA)	Task 33 (RK)
Task 21 (RK)	Task 17 (OA)	Task 8 (RK)	Task 26 (OA)	Task 11 (RK)	Task 34 (RK)
Task 27 (OA) End		Task 9 (OA)		Task 12 (RK)	Task 35 (OA)
				Task 13 (RK)	Task 36 (OA)
				Task 14 (OA)	Task 37 (OA)
				Task 15 (RK)	Task 32 (RK)

Legende:

User Stories	Tasks	Milestones	In Progress	Abge-schlossen

iii. Iteration 3

Iteration 3					
Wochen 7-12					
Code: Erstes Refactoring vor Start Iteration 3	Einlesen in Spieltheorie und Algorithmen	Funktion / Logik: Implementation des Spielbaumes	Funktion: Implementation der Heuristik	Funktion: Implementation Alpha-Beta Algo	Funktion / Logik: Verfeinern der Spielstärke
Task 38 (RK)	Task 57 (both)	Task 44 (OA)	Task 41 (RK)	Task 47 (both)	Task 52 (OA)
Task 39 (OA)	Task 58 (both)	Task 45 (OA)	Task 42 (RK)	Task 48 (both)	Task 53 (RK)
Task 40 (RK)		Task 46 (OA)	Task 43 (RK)	Task 49 (Both)	
				Task 50 (RK)	
				Task 51 (OA)	

Iteration 3	Mileston
Wochen 7-12	e 3
Gesamtprogramm: Beta Tests durch externe Benutzer	Dokumentation: Finalisieren der Dokumentation
Task 54 (both)	Task 56 (both)
Task 55 (both)	

Legende:

User Stories	Tasks	Milestones	In Progress	Abge-schlossen

c. Aufteilung der Aufgaben

Die Aufgaben des Projekts wurden grösstenteils nach den Stärken der Programmierer vorgenommen. Oliver Aeschbacher hat seine Stärken im GUI Programmieren sowie im Bereitstellen guter Datenstrukturen und Threading, René Kamer hat seine Stärken eher in der Algorithmik sowie im mathematischen Teil und Heuristik.
Absichtlich wurden jedoch gewisse Aufgaben auch direkt diesem Ansatz entgegengesetzt verteilt, damit sich auch während des Projekts Lerneffekte und Horizonterweiterungen einstellen konnten. Kleinere Anpassungen waren nach dieser Vergabe noch nötig, damit die Arbeiten ungefähr gleich verteilt werden konnten auf beide Personen.

d. Anpassungen im Verlauf des Projekts

Ein prinzipieller kleiner Lapsus ist uns bei der Planung passiert, was sich im Nachhinein als gut für uns erwiesen hat. In der festen Annahme, dass es drei Iterationsphasen mit je drei Wochen gibt, haben wir die Planung entsprechen ausgeführt. Im Verlauf des Projekts mussten wir jedoch feststellen, dass wir vier Iterationsphasen mit je drei Wochen zur Verfügung hatten. Dies hat uns jedoch gut in die Hände gespielt, da gegen Ende des Semesters der Zeitplan wesentlich straffer wurde und wir pro Woche nicht mehr so viel Zeit in dieses Projekt investieren konnten. Ebenfalls sind in der letzten Iterationsphase noch einige Probleme aufgetaucht. Dazu aber später mehr.
Des Weiteren haben wir den Ablauf noch angepasst, um animierte Spielsteine zu erzeugen. Dies hatten wir aus zwei Gründen getan:

- Man findet keine oder nur sehr wenige Implementationen im Netz, welche animierte Spielsteine implementiert haben. Dies wollten wir ändern.
- Wir haben selbst sehr hohe Ansprüche an Ausführung und Qualität unserer Arbeit und wollten unbedingt noch eine Animation mit reinnehmen.

5. Umgebung des Projekts

Im Folgenden wollen wir die eingesetzte Technologie sowie die Gründe dafür etwas näher beschreiben. Wir haben uns hierbei grösstenteils auf die in den bereits absolvierten Kursen erlernten Werkzeuge abgestützt.

a. Programmwahl

Wir haben dieses Projekt mit der Programmiersprache Java umgesetzt. Die Entscheidung dafür gestaltete sich aus den folgenden Gründen nicht schwer:

- René Kamer hat während seiner Laufbahn schon viele verschiedene Programmiersprachen erlernt, unter anderem Java
- Oliver Aeschbacher setzt diese Sprache in seinem Berufsalltag ein
- Während den ersten zwei Jahren des Studiums konnten wir ein grosses Basiswissen auf diesem Gebiet erarbeiten
- Entsprechend vereinfachten diese Aspekte auch die Zusammenarbeit und den erfolgreichen Abschluss

b. Tools

Als Entwicklungsumgebung haben wir Eclipse eingesetzt. Diese war uns beiden schon bekannt und hat sich auch in anderen Projekten schon bewährt.

Des Weiteren haben wir die Configuration-Library aus den Apache Commons Projekten verwendet. Diese hat sich für die Spieleinstellungen als sehr nützlich erwiesen. Die Einstellungen werden in einem Properties-File abgespeichert. Der Vorteil der Configuration-Library war, dass beim Speichern

– im Gegensatz zur Java-API – die Kommentare und die Reihenfolge der Einträge nicht verloren gehen.

Als letztes ist noch Git als Versionierungswerkzeug zu nennen. Die zentrale Verwaltung unseres Codes und der Dokumentationen haben wir über Github abgewickelt.

6. Reversi

a. Geschichte des Spiels

Im Internet werden mehrheitlich zwei Versionen zur Entstehung beschrieben. Eine davon lässt den Ursprung auf Japan unter dem Namen „Othello“ in den frühen 70ern schliessen. Die andere behauptet, die Entstehung sei in England im Jahr 1888 gewesen. Diese Version wird vielfach vermerkt, deshalb werden wir uns im Folgenden darauf beziehen.

1888 hat Lewis Waterman das Spiel unter dem Namen „Reversi“ erfunden. Allerdings soll John W. Moller bereits den Grundstein für die Idee gelegt haben. Der Spielname stammt aus dem Lateinischen und bedeutet „umkehrbar“. Die Firma Ravensburger vermarktete 1893 dieses Brettspiel und so wurde es eines der erfolgreichsten Spiele der Neuzeit, welches bis heute Spieletheoretiker auf der ganzen Welt beschäftigt.

Othello und Reversi sind eng Verwandte Brettspiele und unterscheiden sich nur in sehr wenigen Punkten (im Folgenden Abschnitt genauer erklärt). Insbesondere bei den elektronischen Varianten wird der Begriff heute mehrheitlich synonym verwendet.

b. Regeln / Erklärung des Spiels

Reversi bzw. Othello ist ein Brettspiel für zwei Personen, welches mit weissen und schwarzen Steinen gespielt wird. Ziel des Spiels ist es, möglichst viele Steine seiner Farbe auf dem Brett liegen zu haben.

Das Spielbrett besteht wie beim Schach aus 8 x 8 Feldern. Der Unterschied zum Schach ist, dass die Feldnummern von oben nach unten geschrieben werden.

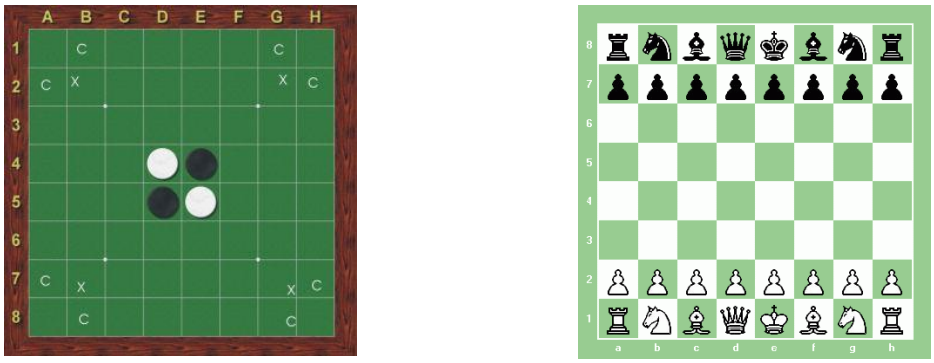


Abbildung 1: Reversi vs Schach

Die Grundaufstellung ist zwischen Othello und Reversi verschieden: Bei Othello ist dies wie in Abbildung 1 ersichtlich gegeben. Es liegen zwei weisse und zwei schwarze Steine. Bei Reversi werden diese Steine in den ersten Zügen gesetzt. Somit kann die Eröffnung variieren.

Dann wird abwechselnd gespielt. Der aktuelle Spieler setzt einen seiner Steine auf ein freies Feld. Der Zug muss so durchgeführt werden, dass mindestens ein gegnerischer Stein zwischen den eigenen

eingeschlossen wird. Dies kann in beliebiger Richtung erfolgen (waagrecht, senkrecht oder diagonal). Die eingeschlossenen gegnerischen Spielsteine werden anschliessend umgedreht bzw. durch Steine der eigenen Spielfarbe ersetzt.

Beispiel (Weiss ist am Zug):

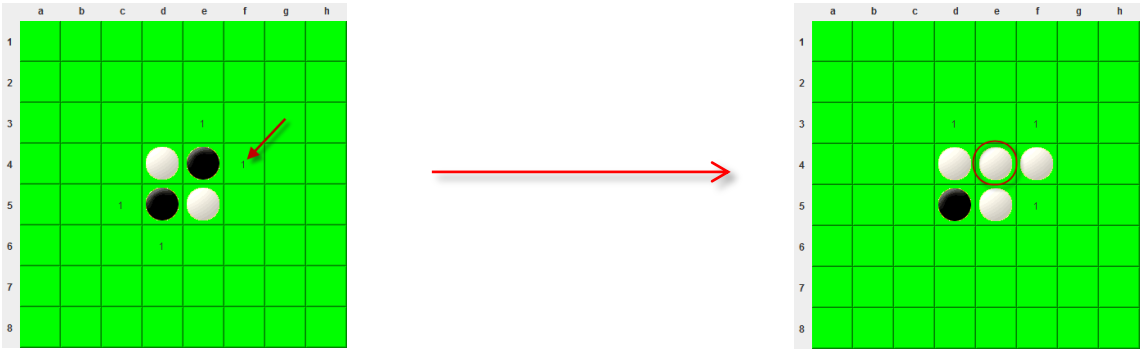


Abbildung 2: Erklärung: Wie zieht man

Weiss setzt den Stein auf f4. Dadurch wird der schwarze Stein auf e4 „geschlagen“.

So wird das Spiel fortgesetzt bis entweder das Spielbrett voll ist oder nur noch Steine einer Spielfarbe auf dem Brett liegen. Gewonnen hat derjenige mit den meisten Steinen. Falls beide Spieler gleichviele Steine haben, ergibt dies ein unentschieden.

In gewissen Situationen ist es möglich, dass man keinen Stein legen kann, weil keine gegnerischen Steine eingeschlossen werden können. Dann muss „gepasst“ werden und der gegnerische Spieler ist wieder am Zug. Falls nun dieser Spieler ebenfalls passen muss, ist das Spiel beendet.

c. Aktuelle Geschehnisse

Im Internet ist dieses Spiel in grosser Anzahl vertreten. Es wird auf verschiedenen Online-Plattformen angeboten oder steht zum Download bereit. Hierzu findet man in nahezu jeder Programmiersprache und für jede Plattform eine Version.

Das Spiel hat sich aber auch als Brettspiel sehr weit verbreitet. Seit 1977 findet jährlich eine Weltmeisterschaft statt.

Besonders die Othello-Variante wir im asiatischen Raum sehr intensiv ausgeübt. Beispielsweise finden in Singapur sogar Schulmeisterschaften statt.

7. Spieletheorie

a. Spielbaum-Theorie allgemein

Vorbemerkung: Ein Spielbaum lässt sich für beinahe jedes Spiel aufstellen, wie z.B. für Schach, Reversi, TicTacToe oder VierGewinnt.
Wir betrachten ein sehr einfaches Spiel, in dem es in jeder Spielstellung nur zwei erlaubte Züge gibt, links (l) oder rechts (r). Befindet sich das Spiel nun in einer bestimmten Spielstellung S und Spieler A führt einen Zug aus, so sind zwei Folgestellungen möglich, S_l und S_r. Abbildung 3 veranschaulicht diese Situation durch einen gerichteten Graphen. Die Knoten des Graphen entsprechen den Spielstellungen die (von oben nach unten gerichteten) Kanten des Graphen entsprechen den Spielzügen. Der Graph ist ein Baum, der Spielbaum für einen Zug ausgehend von Spielstellung S.

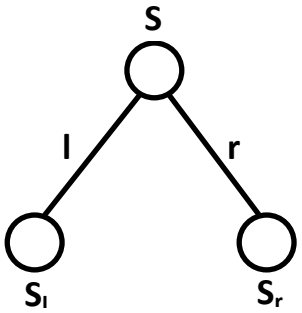


Abbildung 4: Spielbaum 1

Danach ist Spieler B am Zug. Je nachdem, ob er Spielstellung S_l oder S_r vorfindet und je nachdem, ob er Zug l oder Zug r ausführt, ergibt sich anschließend eine der vier möglichen Spielstellungen S_{ll}, S_{lr}, S_{rl}, S_{rr}. Abbildung 5 veranschaulicht diese Situation wieder anhand des entsprechenden Spielbaums. Nach einem weiteren Zug von Spieler A und einem weiteren Zug von Spieler B sind insgesamt 16 Spielstellungen möglich. Es ist möglich, dass unterschiedliche Zugfolgen zu der gleichen Spielstellung führen. D.h. unterschiedliche Knoten des Spielbaums können gleiche Spielstellungen repräsentieren.

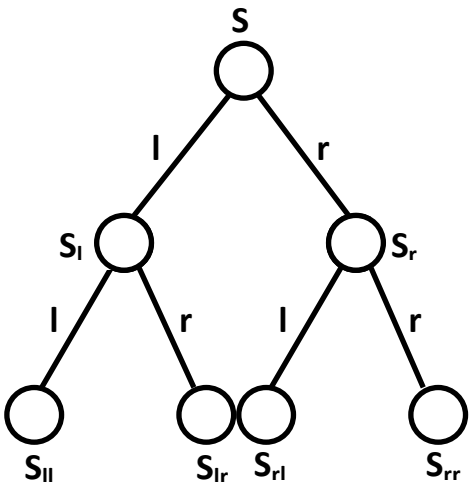


Abbildung 5: Spielbaum 2

b. Spielbaum-Theorie Reversi

Übertragen auf das Spiel Reversi kann man nun sagen, dass es nicht nur zwei erlaubte Züge geben kann, sondern sehr viele (um herauszufinden, was das Maximum an möglichen Spielzügen ist, müsste man alle möglichen Spielsituationen durchrechnen). Dargestellt aus der Startsituation stellt sich der Spielbaum beim Reversi so dar:

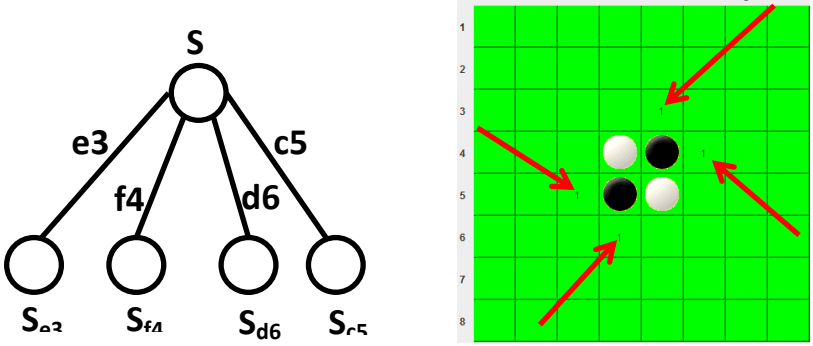


Abbildung 6: Spielbaum Reversi

Entsprechend verläuft auch die Fortsetzung des Spielbaums wesentlich breiter, auf eine entsprechende Grafik wird hier verzichtet.
Wie man sich vorstellen kann, wird ein Reversi-Spielbaum mit zunehmender Spieldauer sehr breit werden (12-13 mögliche Züge sind keine Seltenheit in Spielsituationen) und die Datenrepräsentation wird entsprechend sehr viel Speicher benötigen.

c. Heuristik allgemein

Das Ergebnis der folgenden Algorithmen hängt massgeblich von der Bewertungsfunktion, welche in den meisten Fällen eine Heuristik ist, ab, daher werfen wir hier einen genaueren Blick darauf.
Eine Bewertungsfunktion ordnet jedem Zustand p einen Wert x zu über die Funktion f(p), wobei der Zustand den jeweiligen Brettsituationen in den Blättern des Baumes entspricht. Dies kann eine sehr einfache oder auch sehr komplexe Heuristik bzw. Funktion sein.
Eine ideale Bewertungsfunktion ordnet einer Stellung den Wert +1 zu, wenn Spieler A gewinnt und den Wert -1, wenn B gewinnt. Bei Unentschieden entsprechend den Wert 0.
Kann man den Spielbaum vollständig aufbauen, spielt der Algorithmus ein perfektes Spiel, was in der Realität jedoch nur bei sehr einfachen Spielen wie z.B. TicTacToe möglich ist.
Bei allen anderen Spielen modifiziert man die Bewertungsfunktion, aber immer noch so, dass Spieler A bei positiver Aussicht eine positive Bewertung hat und umgekehrt.

d. Heuristik Reversi

Bei Reversi sind verschiedene Bewertungsfunktionen möglich. Es hat sich aber eine allgemeine, auch für Anfänger verständliche Heuristik durchgesetzt, welche hier im Detail anhand einer Graphik erklärt wird. Diese Erklärung bezieht sich hauptsächlich auf das Mittelspiel, da die Erklärungen zu Eröffnung und Endspiel den Rahmen dieser Abhandlung sprengen würde.

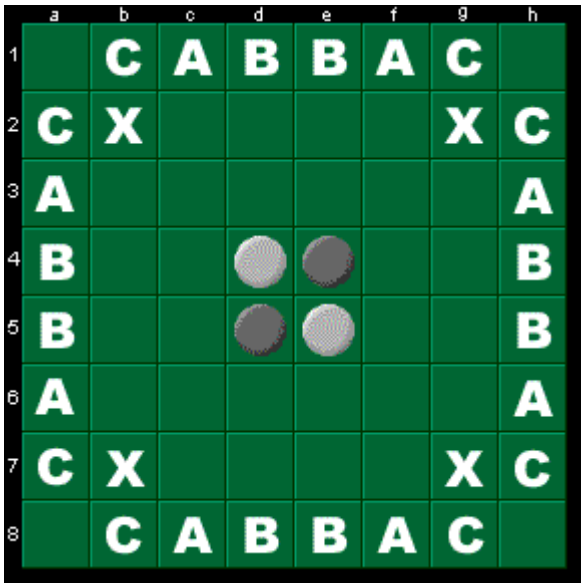


Abbildung 7: Heuristik Reversi

- Generell empfiehlt es sich, so wenige Spielsteine wie möglich umzudrehen. Dies hat den Hintergrund, dass je weniger Steine man hat, umso weniger kann der Gegner schlagen. Des Weiteren erhöht eine kleine Anzahl von Steinen die Zugmöglichkeiten enorm (Fachbegriff: Mobility & Agility)
- Solange kein triftiger Grund besteht, sollte man in den „Sweet Sixteen“ bleiben (die 16 Felder im Zentrum des Spielfeldes). Der Grund: sobald man diese Verlässt, betritt man ein sogenannt „heisses“ Feld.
- Wenn man als erster einen Spielstein an den Rändern platziert, ist es im generellen besser auf ein A oder B Feld zu spielen als auf ein C Feld. Der Grund ist, dass man beim C-Feld, wenn es geschlagen wird, die Ecke verliert (Ecken können nicht geschlagen werden und haben enorme Macht)
- Man sollte verhindern, eine „Wand“ zu bilden (eine zusammenhängende Linie von Spielsteinen am Rand der Spielsteine, welche nicht am Rand des Feldes liegen). Hintergrund: diese Spielsteine bergen grosse Gefahr und haben geringe Mobilität
- Wenn der Gegner eine „Wand“ bildet, sollte man solange wie möglich NICHT durch diese hindurchbrechen. Hintergrund: Aufgeschoben ist nicht aufgehoben.
- Es ist immer besser, wenn man die Kontrolle über Spielsteine hat, wenn diese quadratisch in der Mitte einer Gruppe von Steinen liegen, als Grenzsteine (am Rand der bestehenden Spielsteine) zu haben. Der Grund liegt abermals in der Mobilität und Agilität.

Wenn man diese Regeln in der Heuristik abbilden kann (was wirklich nicht einfach ist), wird die Heuristik relativ gute Resultate liefern.

e. MiniMax Algorithmus

Mit der Theorie der Spielbäume und dazugehörend der Theorie der Bewertungsfunktionen können wir nun eine Vorversion des Alpha-Beta Algorithmus betrachten: den MiniMax Algorithmus. Dieser ist anwendbar (wie der Alpha-Beta Algorithmus auch) auf Spielbäume aller Art.
In untenstehender Grafik ist ein solcher beliebiger Spielbaum aufgezeigt, von der zugrundeliegenden Spielsituation 4 Züge in die Tiefe (entspricht logischerweise der Suchtiefe 4)

In der Wurzel (Kopfknoten) ist A am Zug, A spielt also die Züge 0 und 2. Hier wird jeweils der untergeordnete Zug maximiert, also der für Spieler A günstigste Zug ausgewählt.
Die Knoten bei 1 und 3 entsprechen Situationen, in denen B am Zug ist, hier wählt B den für sich günstigsten Zug. Da wir jedoch in diesem Beispiel für A rechnen, müssen diese Knoten minimiert werden.
Angefangen bei den Blättern arbeitet sich der Algorithmus so hoch bis zu Wurzel. Der Wurzel wird also von seinen untergeordneten Knoten der grösste Wert zugewiesen und dieser Knoten wird dann auch gespielt (in der Graphik der Wert -7, was einem spezifischen Zug entspricht)

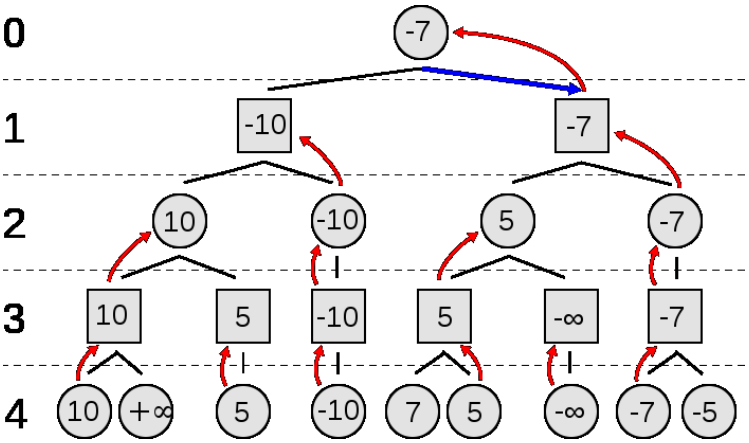


Abbildung 8: Minimax Algorithmus

f. Alpha-Beta Algorithmus

Der Alpha-Beta Algorithmus ist ein direkter Verwandter des Minimax Algorithmus und beruht auf dessen Grundprinzip des Minimierens oder Maximierens. Jedoch spart der Alpha-Beta Algorithmus, bei GLEICHER Funktionalität, eine Vielzahl der Suchschritte (ein kurzer Vergleich findet sich im nächsten Unterabschnitt). Er ist ein rekursiver Algorithmus, welcher die sogenannte Tiefensuche implementiert, d.h. einen Spielbaum von links nach rechts durchläuft.
Zuerst eine kurze Begriffseinführung:

- Ein Zug, bei dem Spieler A am Zug ist (z.B. bei der Wurzel) nennen wir MAX-Stellung
- Ein Zug, bei dem Spieler B am Zug ist (z.B. direkt nach der Wurzel) nennen wir MIN-Stellung

Das Prinzip ist folgendermassen:

- Man stelle sich einen Zettel vor, auf dem immer zwei Zahlen [alpha, beta] stehen. Interessant ist die übergebene Zahl nur dann, wenn sie zwischen diesen zwei Zahlen liegt. Ein Beispiel: Wenn auf dem Zettelchen [-5, 2] und der übergebene Wert -1 ist, so wird diese Zahl übernommen. Falls er aber z.B. 6 ist, wird er nicht übernommen (liegt ausserhalb des Fensters).
- Schreibe alle möglichen Spielzüge, die in der entsprechenden Spielstellung möglich sind auf ein Stück Papier. Bearbeite alle Züge auf der Liste nach folgender Weise:
- Falls wir in einer MAX-Stellung sind*:

    - Werte den folgenden Zug aus (dies kann auch bedeuten, dass wir in die nächste Ebene, also eine MIN-Stellung kommen), wenn der Rückgabewert grösser ist als alpha, übermale alpha mit diesem Rückgabewert..
    - Falls die neue Zahl alpha grösser oder gleich der Zahl beta wird, gib die Zahl alpha eine Stelle nach oben und schaue keine neuen Züge mehr an. Falls die Zahl alpha immer noch kleiner ist als beta, nimm den nächsten Zug auf der Liste und gehe zurück zu *.
- Falls wir in einer MIN-Stellung sind**:
    - Werte den folgenden Zug aus (dies kann auch bedeuten, dass wir in die nächste Ebene, also eine MAX-Stellung kommen), wenn der Rückgabewert kleiner ist als beta, übermale beta mit diesem Rückgabewert.
    - Falls die neue Zahl beta kleiner oder gleich der Zahl alpha ist, gib beta eine Stelle nach oben und schaue keine neuen Züge mehr an. Falls die Zahl beta immer noch grösser ist als alpha, nimm den nächsten Zug auf der Liste und gehe zurück zu **.
- An diese Stelle kommen wir nur, wenn für alle Züge der Liste Ergebnisse von unten im Spielbaum gekommen sind. Wenn wir in einer MAX-Stellung sind, gib die Zahl alpha zurück. Falls wir in einer MIN Stellung sind, gib beta zurück. Dieser Wert bestimmt den jeweils besten Zug.

Machen wir ein Beispiel mit jeweils zwei möglichen Spielzügen (z.B. links / rechts) (Quadrate sind MAX-Stellungen, Kreise MIN-Stellungen):

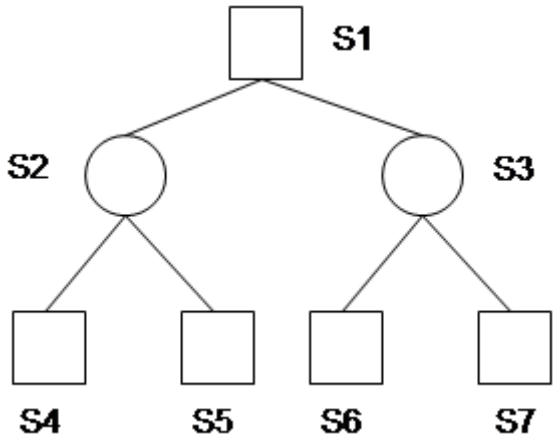


Abbildung 9: AlphaBeta 1

Die Zahlen alpha und beta werden initialisiert mit [-∞, +∞]. Der Algorithmus läuft zuerst links herunter bis zu S4 (Gemäss informellem Algorithmus sind wir nun auf dem Blatt und können die Spielsituation bewerten). Die Bewertung ergibt einen Wert von 4 für S4.

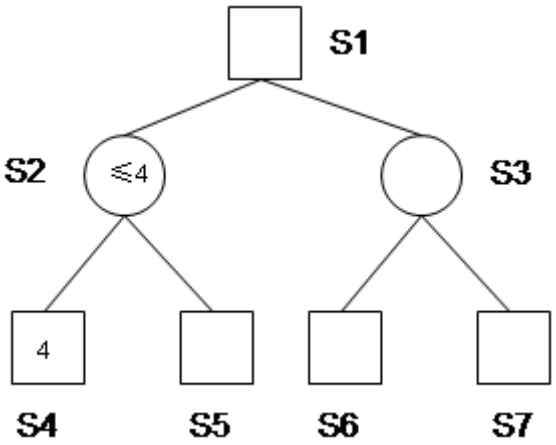


Abbildung 10: AlphaBeta 2

Für S2 wissen wir nun, dass der Wert von S2 kleiner oder gleich 4 sein muss, da der MIN Spieler einen Nachfolger mit minimalem Wert wählt. Danach geht es weiter bei S5, wo wir als Bewertung eine 3 bekommen. Zurück in Stellung S1 wissen wir, dass der Wert von S3 grösser oder gleich 3 ist. Denn wenn der MAX Spieler nach links geht, wissen wir, dass er den Wert 3 erreicht. Die rechte Hälfte ist noch nicht ausgewertet, der MAX Spieler in S1 „sieht“ den Wert noch nicht.

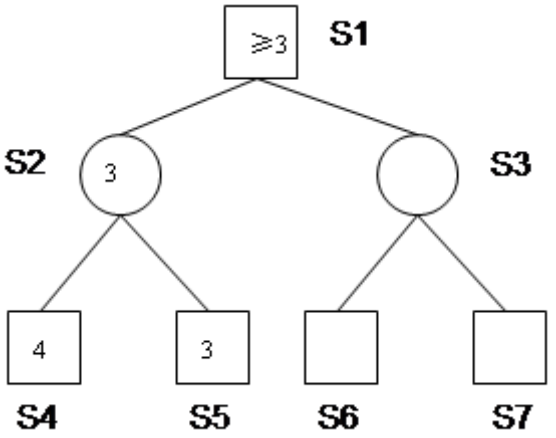


Abbildung 11: AlphaBeta 3

Jetzt kommt des Algorithmus grosse Stunde. Wenn wir nun für Stellung S3 auswerten, werden die Zahlen alpha und beta benötigt. Die Stellungen unterhalb von S3 werden nun nur interessant, wenn von S3 grösser als 3 ist. Das Zahlenpaar [3, ∞] wird nun an S3 gegeben. Alpha ist nun nicht grösser oder gleich beta, also wird S3 ausgewertet.

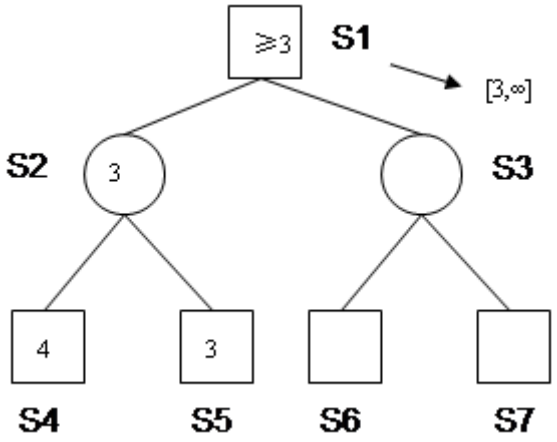


Abbildung 12: AlphaBeta 4

Folglich wird nun S6 ausgewertet und es wird z.B. der Wert 2 gefunden. Dieser wird nun an S3 hochgegeben und dieser setzt beta nun auf 2. Jetzt ist im Spielzug S3 bekannt, dass der Wert von S3 höchstens 2 ist und dass der genaue Wert von S3 in S1 nur von Interesse wäre, wenn der Wert grösser als 3 ist.

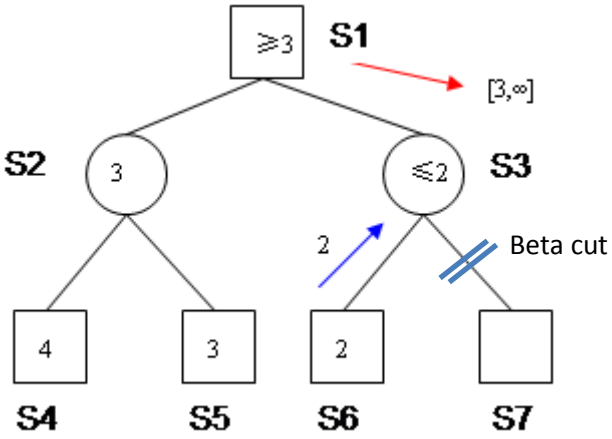


Abbildung 13: AlphaBeta 5

Das heisst aber, dass egal was in S7 für ein Wert gefunden würde, die Stellung S3 die Entscheidung des MAX-Spielers in S1 nicht mehr beeinflussen würde. Daher wird S7 auch nicht mehr untersucht und wird „weggeschnitten“.

Nach dieser ausführlichen Erklärung überführen wir unseren Algorithmus in eine etwas gängigere Repräsentation:

```
int alphabeta(Node v, int alpha, int beta){
      if(v ist Blatt){
            return Bewertung;
      }
      for each child w of v do{
            if(MAX-Spieler ist am Zug){
                  alpha = MAX(alpha, alphabeta(w, alpha, beta));
                  if(alpha >= beta){
                        return alpha;
```

```
                }
            }
            else{
                beta = MIN(beta, alphabeta(w, alpha, beta));
                if(alpha >= beta){
                    return beta;
                }
            }
        }
        if(MAX-Spieler ist am Zug){
            return alpha;
        }
        else{
            return beta;
        }
}
```

Es existieren einige Verbesserungen des Algorithmus, welche jedoch hier nicht im Einzelnen erklärt werden. Jedoch möchten wir sie hier kurz erwähnen, damit der Leser dieses Dokuments diese auch einmal gehört hat:

- Vorsortierung der Züge
- Principal Variation Suche
- Iterative Tiefensuche
- Killer Heuristik
- Quiescent Suche
- Null Zug Suche

g. Vergleich der Minimax-Suche mit der Alpha-Beta Suche

Um kurz aufzuzeigen, wie man Züge spart, wird hier noch ein kleiner Vergleich angestellt zwischen Minimax Algorithmus und AlphaBeta Algorithmus. Es handelt sich dabei um Schätzungen, denen Zugrunde liegt, dass die Teilbäume in etwa gleich gross sind:

Algorithmus	Bewertungen	Cutoffs	Anteil der Cutoffs	Rechenzeit in sek.
MiniMax	28.018.531	0	0,00 %	134,87 s
AlphaBeta	2.005.246	136.478	91,50 %	9,88 s

8. Umsetzung des Projekts

Wir haben das GUI auf Basis eines JApplet entwickelt. Auf diese Weise kann das Spiel auch auf einer Webseite eingebunden werden.

a. GUI

i. Darstellung / Aufbau

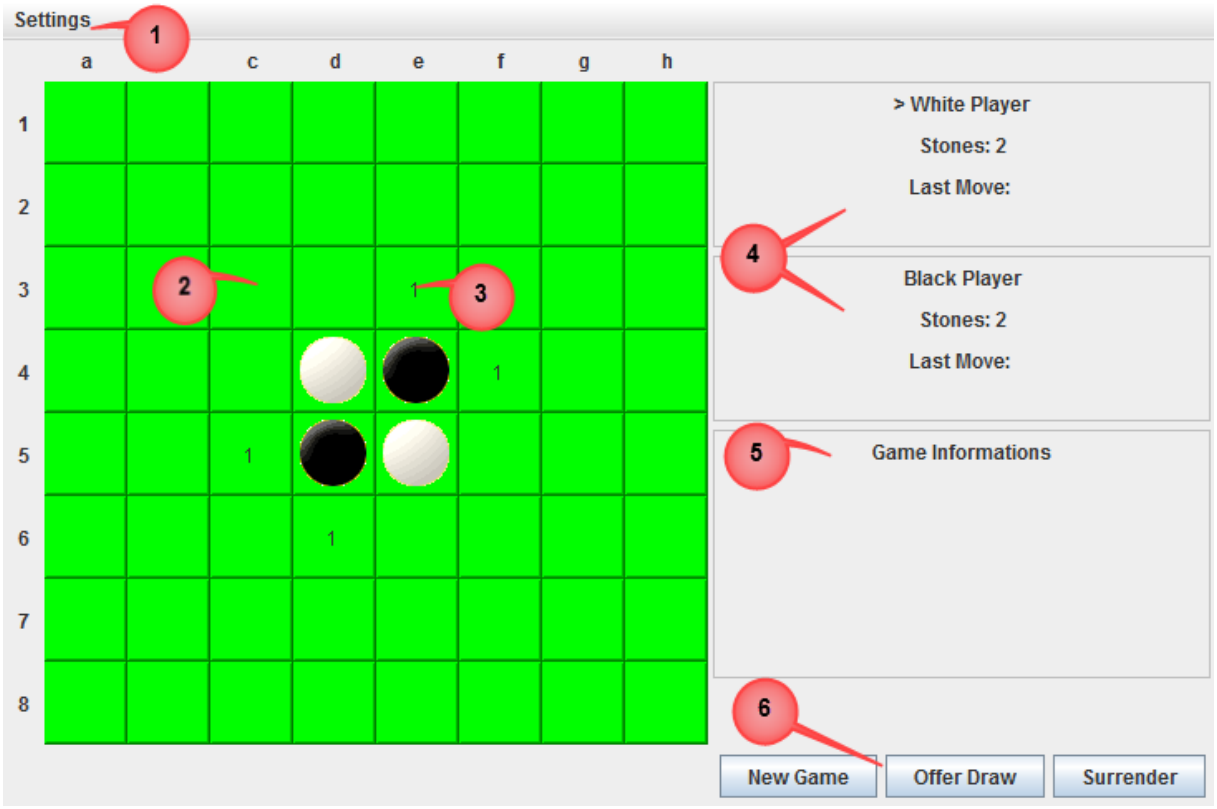


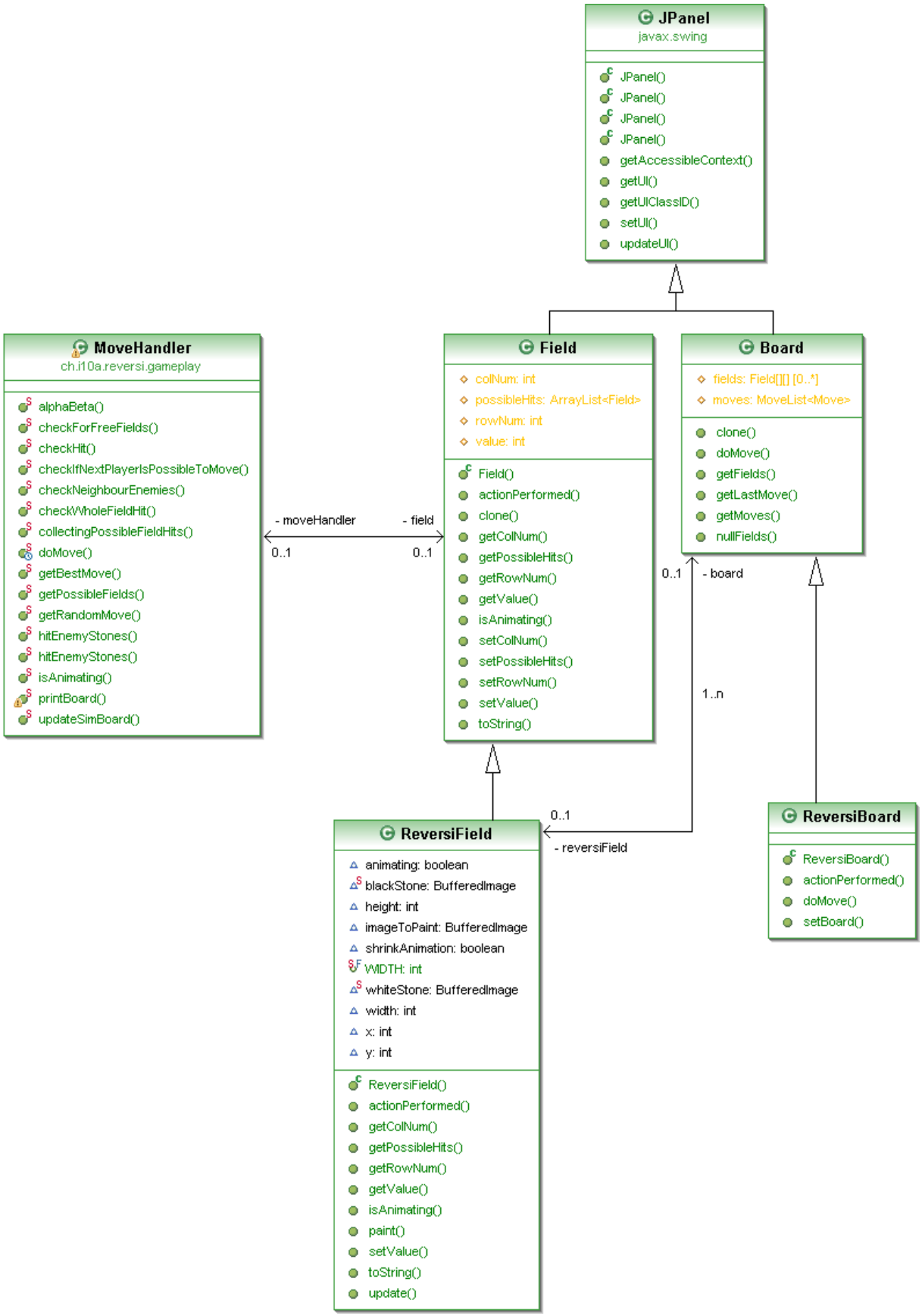
Abbildung 14: Reversi GUI

1. Spieleinstellungen
2. Leeres Feld, auf welches **nicht** gespielt werden darf
3. Leeres Feld, auf welches gespielt werden darf. Die Zahl beschreibt die Anzahl gegnerischer Spielsteine, die umgedreht werden
4. Spieler-Informationen: Name bzw. Spielfarbe, Anzahl Steine, zuletzt getätigter Zug. Der Spieler, welcher am Zug ist, wird mit dem Präfix „>“ dargestellt
5. Allgemeine Spielinformationen: Eröffnung, Spielstati (Gewonnen, unentschieden, …)
6. Schaltflächen um ein neues Spiel zu starten, ein Unentschieden anzubieten oder aufzugeben

ii. Technische Umsetzung

Die einzelnen Komponenten (Board, Felder, Informationsanzeigen, …) bauen auf der Swing-Komponente JPanel auf.

Das nachfolgende UML-Diagramm zeigt das Zusammenspiel zwischen den GUI-Komponenten und der für die Zugdurchführung zentralen Klasse MoveHandler:


JPanel
javax.swing
JPanel()
JPanel()
JPanel()
JPanel()
getAccessibleContext()
getUI()
getUIClassID()
setUI()
updateUI()
MoveHandler
ch.i10a.reversi.gameplay
alphaBeta()
checkForFreeFields()
checkHit()
checkIfNextPlayerIsPossibleToMove()
checkNeighbourEnemies()
checkWholeFieldHit()
collectingPossibleFieldHits()
doMove()
getBestMove()
getPossibleFields()
getRandomMove()
hitEnemyStones()
hitEnemyStones()
isAnimating()
printBoard()
updateSimBoard()
- moveHandler
0..1
- field
0..1
Field
colNum: int
possibleHits: ArrayList<Field>
rowNum: int
value: int
Field()
actionPerformed()
clone()
getColNum()
getPossibleHits()
getRowNum()
getValue()
isAnimating()
setColNum()
setPossibleHits()
setRowNum()
setValue()
toString()
Board
fields: Field[][] [0..*]
moves: MoveList<Move>
clone()
doMove()
getFields()
getLastMove()
getMoves()
nullFields()
0..1
- board
1..n
0..1
- reversiField
ReversiField
animating: boolean
blackStone: BufferedImage
height: int
imageToPaint: BufferedImage
shrinkAnimation: boolean
WIDTH: int
whiteStone: BufferedImage
width: int
x: int
y: int
ReversiField()
actionPerformed()
getColNum()
getPossibleHits()
getRowNum()
getValue()
isAnimating()
paint()
setValue()
toString()
update()
ReversiBoard
ReversiBoard()
actionPerformed()
doMove()
setBoard()

iii. Update Problematik

Im Verlaufe des Projektes hatten wir mit mehreren Problem zu kämpfen. Unter anderem, dass die Aktualisierung von Spielinformationen, welche über die Paint-Methode von Java getätigt wurde, zu spät angezeigt wurden.

Als Beispiel:

Anfangs hatten wir den Spielstein (Weiss / Schwarz) bei der jeweiligen Spieler-Information angezeigt. Also wurde – wenn Weiss am Zug war – ein weisser Stein neben dem Spielernamen angezeigt. Sobald Weiss gezogen hatte, wurde beim schwarzen Spieler ein schwarzer Stein angezeigt. Dies hat jedoch nach der Implementation des Computers nicht mehr funktioniert bzw. wurde zu spät durchgeführt.

Das Problem haben wir gelöst, in dem wir die Anzeige angepasst haben. Wir setzen die Informationen über JLabels und verzichten auf die grafische Anzeige des Spielsteins.

iv. Thread Problematik

Wir hatten schon sehr früh die Ambition, das Umdrehen der Steine zu visualisieren. Dafür haben wir die Animation in einem separaten Thread durchgeführt. Jedoch hat auch dieses Feature nur solange funktioniert, bis wir die K.I. implementiert haben. Die Animation lief asynchron, d. h. dass wenn der letzte Zug gerade noch animiert wurde, hat der Computer-Spieler seinen Zug bereits ausgeführt, was zu inkonsistentem Zustand des Spielbretts führte.

Dies konnte gelöst werden, indem wir Swing-Timers eingesetzt haben. Die Animationen werden über diese Komponente ausgeführt, der gesamte Ablauf des Spielzuges ist aber dennoch synchron. Besten Dank an dieser Stelle an unseren Klassenkameraden Tobias Hermann, der uns den entscheidenden Hinweis gegeben hat.

b. Traversieren der Strukturen

Das Board enthält ein zweidimensionales Array mit den Spielfeldern. Damit die gültigen Züge ermittelt werden können, haben wir Algorithmen für die Abfrage entwickelt:

Für ein bestimmtes Feld wird geprüft, ob ein gültiger Zug ausgeführt werden kann. Dies erreichen wir, in dem wir alle benachbarten Felder auf gegnerische Spielsteine überprüfen:

Beispiel für eine senkrechte Überprüfung von oben nach unten:

```
/**
 * Returns true if hit(s) are possible towards the bottom.
 * Example (a is the active field, r is the place the enemy is, i is where your stone is):
 * |-----|-----|-----|
 * |     |     |     |
 * |-----|-----|-----|
 * |     |  a  |     |
 * |-----|-----|-----|
 * |     |  r  |     |
 * |-----|-----|-----|
 * |     |  r  |     |
 * |-----|-----|-----|
 * |     |  i  |     |
 * |-----|-----|-----|
 * @return false if no stones can be hitten
 */
private static boolean checkBottomHit(Field field, PlayerI player){
        boolean hit = false;
        if(checkNeighbourEnemiesBottom(field, player)){
                while(checkNeighbourEnemiesBottom(field, player)){
                        field = getNeighbourBottom(field);
                }
                field = getNeighbourBottom(field);
                if (field == null) {
                        return false;
                }
                if (field.getValue() == player.getValue()){
                        hit = true;
                }
        }
        return hit;
}


/**
 * Returns the neighbour bottom to the active field. If
 * the field is at the bottom border of the board, this method will
 * return null.
 * Example (a is the active field, r is the one to be returned):
 * |-----|-----|-----|
 * |     |     |     |
 * |-----|-----|-----|
 * |     |  a  |     |
 * |-----|-----|-----|
 * |     |  r  |     |
 * |-----|-----|-----|
 * @return null if field falls out of board, the corresponding field otherwise
 */
private static Field getNeighbourBottom(Field field) {
        int activeFieldRowNum = field.getRowNum();
        if (activeFieldRowNum == 7) {
                return null;
        }
        return fields[field.getColNum()][activeFieldRowNum + 1];
}
```

Die Art und Weise der Überprüfung in andere Richtungen sind im Grundsatz identisch.

c. Auslesen der Spielsituationen

Der Computer muss auf Grund der momentanen Spielsituation alle möglichen Züge berechnen können. Dies geschieht, indem eine Baumstruktur über alle möglichen Züge aufgebaut wird. Um die aktuelle Spielsituation beizubehalten, gab es grundsätzlich zwei Möglichkeiten für die Realisierung:

1. Den jeweiligen Zug durchführen und anschliessend wieder rückgängig machen
2. Die aktuelle Spielsituation klonen und den Zug durchführen

Beide Varianten haben Vor- und Nachteile:

Das Problem der ersten Variante ist, dass dies schwierig ist, zuverlässig durchzuführen, da dies nur nach dem Abarbeiten eines ganzen Baumzweiges gemacht werden darf. Vorteil ist der geringe Speicherverbrauch.

Die zweite Variante ist einfach zu implementieren, jedoch ist der Speicherverbrauch viel höher, da jeder Knoten eine statische Spielsituation benötigt, d.h. geklont werden muss.

Wir haben uns auf Grund der hohen Gesamtkomplexität für die zweite Variante entschieden. Dadurch konnten wir den Algorithmus für diese Analyse einfacher halten, mussten jedoch in der Spieltiefe Abstriche machen.

d. Heuristik

Wir haben in diesem Projekt drei verschiedene Arten von Bewertungen implementiert. Jede davon repräsentiert eine Schwierigkeitsstufe.

Easy

Diese Bewertungsfunktion sucht sich einen der möglichen Züge zufällig aus.

```
/**
 * gives back a field that is played by the computer.
 * for the easiest difficulty, this is a random field out of the playable ones
 *
 *
 * @return Field that is played randomly
 */
public static Field getRandomMove() {
        List<Field> possibleFields = getPossibleFields(PlayerManager.getActivePlayer());
        Random randomGenerator = new Random();

        if(possibleFields.size()>0){
                return possibleFields.get(randomGenerator.nextInt(possibleFields.size()));
        }
        return null;
}
```

Medium

Hier bewerten wir den Zug, indem wir die Anzahl der umzudrehenden Steine des Gegners

aufsummieren bzw. die eigenen Steine, die geschlagen werden könnten, negativ aufsummieren. Also wird der Zug gespielt, welcher am meisten Treffer landet.

```
/**
 * Here, the calculation for the difficulty medium takes place. mainly, it just counts the black and
 * white stones and adds them up against each other (go for the most possible stones of his own)
 *
 * @param the board that has to be calculated
 *
 * @return integer value that shows how good the prognose is for the given board
 */
private static int calculateSituationMediumStrength(Board board){
        int situationValue = 0;
        Field field[][] = board.getFields();

        for (int i = 0; i < 8; i++) {
                for (int j = 0; j < 8; j++) {
                        if(field[j][i].getValue() == 1){
                                situationValue++;
                        }
                        else if(field[j][i].getValue() == -1){
                                situationValue--;
                        }
                }
        }
        return situationValue;
}
```

Hard

Auf dieser Stufe unterscheiden wir zwischen Anfangs-, Mittel- und Endspiel.

Im Anfangsspiel achten wir darauf, dass möglichst wenige Steine umgedreht werden um die Agilität zu bewahren. Dies erreichen wir, in dem wir den Algorithmus der Stufe Medium verwenden und den zurückgelieferten Wert negieren. Zusätzlich prüfen wir noch, ob es dem Gegenspieler möglich ist, alle unsere Steine zu ergattern. Falls dies der Fall ist, wird der Zug mit einem sehr hohen Faktor negativ bewertet, damit dieser Spielzug in der Alpha-Beta-Auswertung untergeht.

Im Mittelspiel bewerten wir alle Spezialfelder mit gewichteten Faktoren. Zu diesen Feldern gehören die Ecken, die direkten Nachbarn der Ecken (also die C- und X-Felder) und die Randfelder (A- bzw. B-Felder).

Im Endspiel ist es wiederum wichtig, möglichst viele Steine zu gewinnen. Hier kommt ebenfalls wieder die Heuristik der Stufe Medium zum Zuge. Zusätzlich erhöhen wir die Spieltiefe, sobald nur noch zehn Felder zu spielen sind, damit möglichst die Besten Züge gewählt werden.

Diese Heuristik ist – abgesehen von dem Medium-Algorithmus – sehr kompliziert und aufwendig, weswegen wir hier auf den Code im Anhang verweisen (MoveHandler.calculateSituationHardStrength(…))

e. Umsetzung des Alpha-Beta Algorithmus

Die Implementierung des Alpha-Beta-Algorithmus lässt sich in seinem Grundsatz aus dem Pseudocode übernehmen. Das Problem ist allerdings, dass man die Durchführung des Zuges und die Berechnung der möglichen weiteren Züge am richtigen Ort implementiert. Des Weiteren hatten wir das Problem, dass der erste Pseudocode, den verwendet haben, die Iteration innerhalb der Spielerabfrage gemacht hat. Dies führte dazu, dass nach der Auswertung des ersten Blatts alle weiteren Möglichkeiten abgeschnitten wurden (Code: siehe MoveHandler.alphaBeta(…)).

Eine weitere Frage, die uns am Anfang beschäftigte war, wie starten wir den Algorithmus von der aktuellen Position. Nun, wir haben eine weitere Methode eingeführt (MoveHandler.getBestMove(…)), welche zuerst über alle möglichen Felder der momentanen Situation gespielt werden können, durchläuft und über diese Felder den Algorithmus starten. Somit startet unser Algorithmus eigentlich erst auf der MIN-Stellung. Dies mussten wir im Code berücksichtigen. Hier stellte sich auch noch das Problem, dass wir nicht mit den Initial-Werten von Alpha und Beta rechneten, sondern diese aus den vorherigen Berechnungen übernahmen. Dies führte ebenfalls zu viel zu vielen Cut-Offs.

9. Projektfazit

Wir hatten von Anfang an die Planung gut im Griff und haben die Zeit für die Umsetzung realistisch geschätzt. Die grösseren Herausforderungen und Probleme, die sich uns stellten, konnten wir trotz hohem Aufwand gut meistern. Dies ist zu einem grossen Teil unserer guten Teamarbeit zu verdanken. Wir konnten uns stets darauf verlassen, dass die geplante Arbeit ausgeführt wurde. Da wir weit auseinander wohnen, waren wir häufig auf Telefonate angewiesen, was zum Teil auf Grund mangelnder Erreichbarkeit nicht so gut funktionierte. Dies können wir für zukünftige Projekte sicherlich verbessern. Hier könnte ein fixes Zeitfenster für telefonische Besprechungen Abhilfe schaffen.

Alles in allem hat uns diese Projektarbeit sehr viel Spass bereitet. Vor allem der tiefe Einblick in die Spieltheorie sowie die Umsetzung, wie denn ein Computer „denkt“, haben uns sehr inspiriert und weitergebracht.

10. Danksagungen

Wir danken zuerst unserem Mitschüler Tobias Herrmann, wir wären ohne seinen Tipp zum Threading nicht auf die Lösung gekommen.

Des Weiteren danken wir unseren Beta-Testern für ihren Einsatz:

- Mathias Maly
- Manuel Bitzi
- Sergio Ugolini
- Nicole Bühlmann
- Nadja Weibel

11. Quellen

http://www.iti.fh-flensburg.de/lang/algorithmen/graph/spielbaum.htm#footnote
E. Horowitz, S. Sahni: Algorithmen. Springer (1981)
http://www.iicm.tugraz.at/Teaching/theses/2000/_idb9e_/greif/node7.html
http://www.informatik.uni-kiel.de/fileadmin/arbeitsgruppen/realtime_embedded/teaching/ss08/p-soft/p-soft05.pdf
http://de.wikipedia.org/wiki/Minimax-Algorithmus#Bewertungsfunktion (primär Grafiken)
http://games.yahoo.com/help/rv
http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo19.php
Mark Allen Weiss; Data Structures and Problem Solving Using Java
http://docs.oracle.com/javase/6/docs/api/
http://docs.oracle.com/javase/tutorial/ (besonders für Layouting und Threading)
http://home.datacomm.ch/t_wolf/tw/misc/reversi/html/index.html
http://www.onlinespiele-sammlung.de/othello/about-othello.php
http://www.othellosingapore.com/
http://de.wikipedia.org/wiki/Reversi

12. Abbildungsverzeichnis