Pflichtenheft: Chess JavaFX

Projektbezeichnung	Chess JavaFX
Projektleiter	Stefan Hasler
Erstellt am	16.04.2021
Letzte Änderung am	22.04.2021
Status	[in Bearbeitung]
Aktuelle Version	1.1

1. Einleitung

Der Auftraggeber, Larcher GmbH forderte, ein in Java programmiertes Schachspiel zu erstellen mit einer integrierten K.I. Das Endergebnis soll ein voll funktionsfähiges und benutzerfreundliches Schachspiel sein, welches man entweder lokal zu zweit oder lokal gegen eine K.I spielen kann.

Inhalt

2. Allgemeines

2.1. Teammitglieder

Rolle(n)	Name	E-Mail
Teamleiter	Stefan Hasler	sthasste@bx.fallmerayer.it
Programmierer	Lukas Schatzer	stschluk2@bx.fallmerayer.it
Programmierer	Luca Turin	stturluc@bx.fallmerayer.it
Designer	Alex Unterleitner	stuntale@bx.fallmerayer.it

2.2. Abkürzungen welche im Folgenden Dokument verwendet werden

MVC	Model View Controller
ABP	Alpha Beta Pruning
Alg.	Algorithmus

2.3. Spielregeln

Wir folgen den Spielregeln des internationalen Schachverbandes FIDE.

3. Aufbau

3.1. Systemarchitektur

Es wird versucht ein MVC Pattern zu verwenden, um das Grafische vom Logischen zu trennen. Das bietet den Vorteil, dass man im Model unabhängig vom Grafischen, Züge berechnen kann. Wir haben deshalb zwei Schachbretter pro Spiel, einmal das interne Schachbrett des Models, welche alle Berechnungen durchführt und das Schachbrett in der View, welche nur für die Darstellung des Grafischen verantwortlich ist. Der Controller ist zuständig für die Kommunikation zwischen View und Model. Das Model sollte nie direkt auf das Grafische zugreifen und umgekehrt.

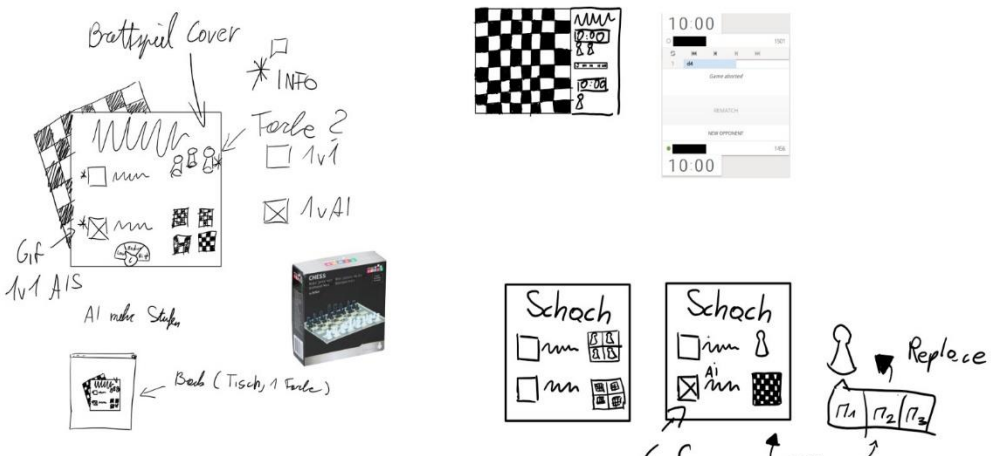


Bild 1 Simples Mockup

Bild 1 stellt unser Mockup für das Menü und das Brett dar, dieses Bild dient nur als grobe Referenz da wir vorhaben es noch zu überarbeiten. Unser Default Brett Design soll dem von Lichess (www.lichess.org) ähneln(Siehe Bild 2).

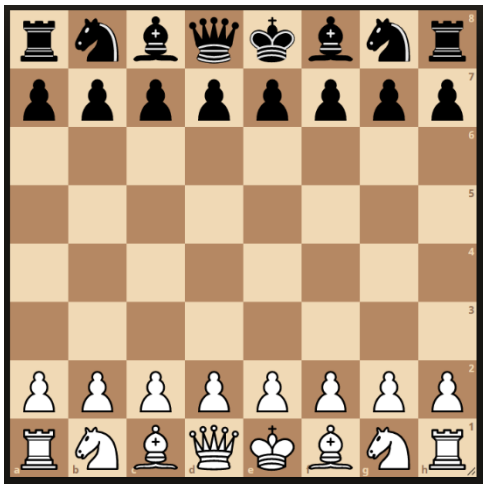


Bild 2 Brett auf Lichess

3.2. Softwarearchitektur

Klassen Diagramm(Siehe Datei KlassenDiagramm.mdj): Dieses Klassendiagramm enthält die Planung des Schachspiels an sich(ohne K.I). Wir haben 3. Packages für das MVC Pattern.

3.2.1. Model

Die Chessboard Klasse soll jeden Zug ein Turn Objekt vom Controller bekommen welche den Zug beschreibt(welches Piece wohin geht), die Chessboard Klasse schaut dann nach ob der Zug legitim ist und gibt ein True oder False dem Controller zurück. Außerdem soll die Chessboard Klasse alle relevanten Informationen über das Spiel speichern(Zuganzahl, alle gespielten Züge, alle Figuren welche sich auf dem Brett befinden, Wer am Zug ist, etc.). Das Chessboard soll dazu noch mehrere Methoden bieten welche zum Manipulieren des Schachbretts dienen, wie z.b.: Schachbrett auf einen Fen setzen, einen Zug zurücknehmen. Das Chessboard selbst wird das Singelton Pattern implementieren, was bedeutet, dass der Konstruktor privat ist und es nur ein Objekt von Chessboard gibt.

Die Field Klasse beschreibt ein Field auf dem Schachbrett sie hat Attribute wie ein x, ein y und einen Namen(„a5“, „e4“, etc.). Auch hat sie eine Referenz zu einem Piece Objekt welche null ist, sollte das Feld leer sein.

Die Piece Klasse ist eine Abstrakte Klasse von welcher jede Figur erbt, jede Figur braucht eine Farbe, einen Namen, einen Wert und ein Feld auf welchem es steht. Auch muss jede Figur eine Methode „getMoves()“ implementieren, welche eine ArrayList aus Feldern zurückgibt. Diese Felder sind allen legitimen Bewegungen der Figur, die getMoves Methode achtet darauf, dass es nur Felder sind, welche für die entsprechende Figur Regelgerecht sind(Keine eigenen Figuren schlagen, etc.). Die Methode überprüft aber nicht ober die Bewegung der Figur den eigenen König in Schach stellen würde, das soll vom Chessboard selber gehandhabt werden.

Die Turn Klasse beschreibt einen Zug, sie besitzt alle Relevanten Informationen welche es braucht um den Zug durchzuführen(welches Piece wohin bewegt wird).

Die Enum Color hat zwei Literals, BLACK und WHITE. Jede Figur braucht eines der beiden damit klar ist zu welchem Spieler die Figur gehört.

Die Enum Gamestate hat Literals welche Information über den momentanen Zustand des Spiels geben. Das ist wichtig fürs Chessboard damit es weiß was gerade passiert. Wir haben hierfür auch ein Observer-Pattern eingebaut damit wir das Schachbrett informieren können wenn sich der Gamestate ändert.

3.2.2. Controller

Der Controller ist für die Kommunikation zuständig. Er implementiert das EventHandler Interface und implementiert das Singelton Pattern. Sobald eine Figur auf der View geklickt wird, kümmert sich der Controller darum. Mit dem Klick auf eine Figur wird diese ausgewählt und der Controller fragt dann das Model um alle Legitimen Züge für diese Figur, der Controller wertet diese dann aus und sagt der View wie sie die Felder darstellen soll(grüner Punkt für legitime Bewegung, Roter Rand wenn es Figur schlagen kann. Roter Rand um den König, wenn Schach). Wenn der Spieler dann auf ein legitimes Feld klickt wird ein neues Turn Objekt erstellt. Das Turn Objekt wird dann weiter ans Model geschickt.

3.2.3. View

Die View hat ein eine eigene Chessboard Klasse (ChessboardView), welche für die Darstellung zuständig ist. Diese muss nach jedem Zug aktualisiert werden, indem man ihr den Fen des Schachbretts übergibt. Das Schachbrett der View enthält ein Gridpane in welcher FieldLabel Objekte gegeben werden

FieldLabel ist View Version der Field Klasse aus dem Model, der Unterschied ist das FieldLabel von der JavaFx Label Klasse erbt. Die FieldLabel Klasse hat dazu noch ein paar Attribute und Methoden für die Darstellung des Feldes.

3.2.4. K.I

Wir das Projekt in zwei Teil-Projekte geteilt. Einmal das Schachspiel an sich und einmal die K.I. Natürlich hatten wir bei der Planung des Schachspiels immer im Hinterkopf das wir es so programmieren müssen das ein Computer auch spielen kann. Deshalb auch das MVC-Pattern, um das Logische vom Graphischen zu trennen.

Wir haben uns natürlich schon über einige Schach K.Is informiert, wir haben uns einige Videos zum Thema angeschaut (Zum Beispiel: Coding Adventure: Chess A.I oder 30 Weird Chess Algorithms). Wir wollen bei unserer Schach K.I Alpha-Beta-Pruning verwenden.

Wir haben ein Flussdiagramm(Siehe Datei FlussDiagramm.mdj) für die K.I erstellt, dieses versucht grob zu zeigen wie die K.I „denken“ soll. Es ist ein grobes Beispiel wie die K.I. implementiert wird.

Minimax Algorithmus

ABP ist eine Optimierte Variante des Minimax Algorithmus.

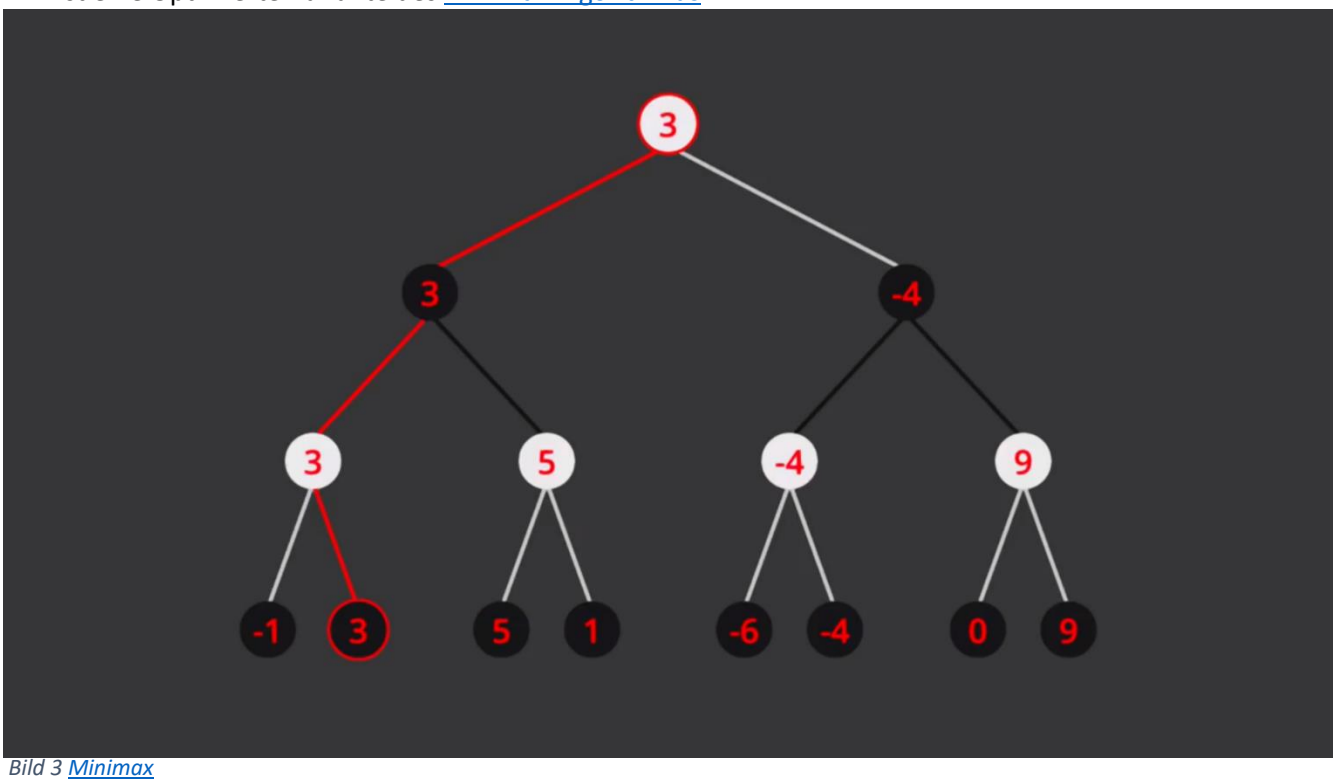


Bild 3 Minimax

Bild 3 Minimax stellt einen Beispiel Baum dar welcher von Alpha Beta Pruning erstellt wird. Um das Beispiel einfach zu halten wird angenommen, dass wir bei jedem Zug nur 2 mögliche Züge haben.

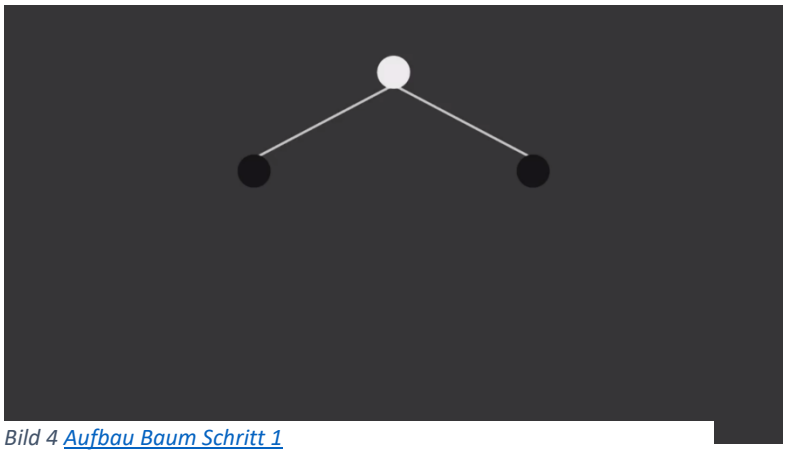
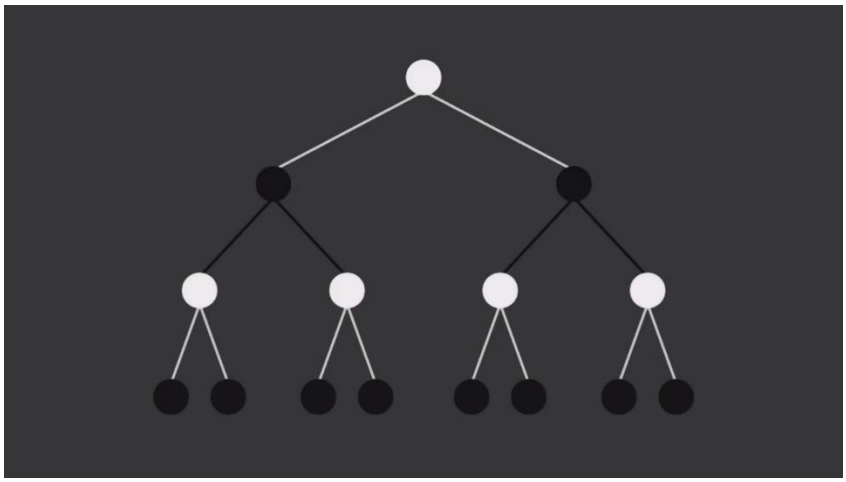


Bild 4 Aufbau Baum Schritt 1

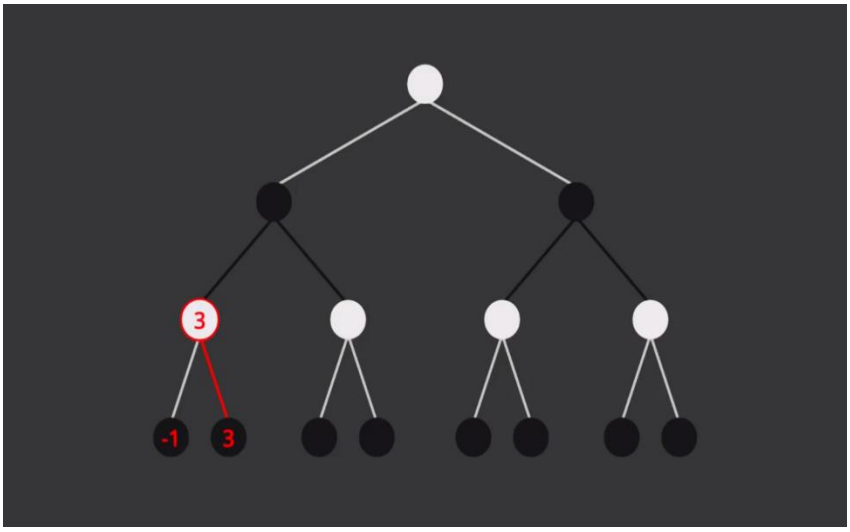
Jeder der Äste repräsentiert einen Zug (siehe Bild 4 Aufbau Baum Schritt 1), nachdem Weiß seinen Zug gemacht hat, ist Schwarz wieder dran.

Die weißen Linien stehen für einen Zug, welcher Weiß durchführt und die schwarzen Linien für schwarze Züge.



Der Baum geht solange weiter bis das Spiel fertig ist oder bis man die gewünschte Tiefe erreicht hat (siehe Bild 5 Baum erstellt).

Bild 5 Baum erstellt



Jetzt muss jedem Node ein Wert zugewiesen werden, der Wert wird beeinflusst durch Dinge wie: Anzahl der schwarzen und weißen Figuren, Position der Figuren auf dem Brett, etc.

Bei der Auswertung will Weiß einen so hohen Wert wie möglich erlangen, Schwarz hingegen will einen so kleinen Wert wie möglichen erhalten.

Deshalb kann man der Node drüber einen Wert von 3 geben, da Weiß den Zug wählen würde, welcher ihm den höchsten Wert gibt.

Bild 6 Werte vergeben

Wertet man den Baum dann vollständig aus, hat man den Baum aus Bild 3 Minimax. Jetzt kann man erkennen, dass Weiß den Zug auf der linken Seite wählen sollte, da er einen Vorteil von +3 hätte(Wenn auch Schwarz immer den besten Zug für sich spielt).

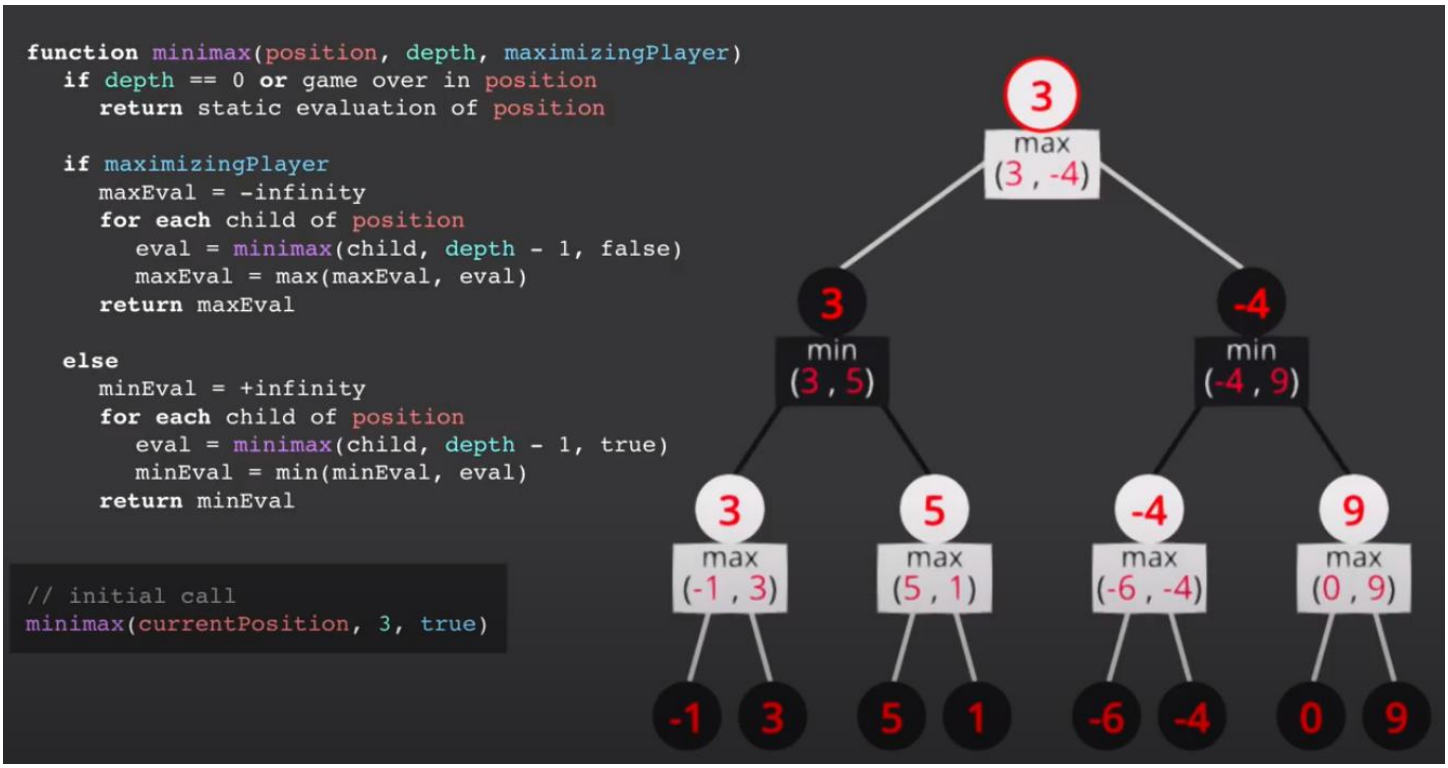


Bild 7 [Pseudocode mit Baum]

Der abgebildete Pseudocode auf Bild 7 Pseudocode mit Baumstellt den Minimax Alg. dar. Die minimax Funktion bekommt die momentane Position, die Tiefe und ein boolean Wert welcher Spieler dran ist übergeben(MaximizingPlayer = Weiß).

Als erstes wird nachgeschaut ob man die gewünschte Tiefe erreicht hat, falls ja wir die Auswertung der Position zurückgegeben. Wenn man die Tiefe noch nicht erreicht hat, wird nachgeschaut welcher Spieler dran ist, mit „for each child of position“ ist gemeint, dass hier jetzt alle Züge nachgeschaut werden, die der Spieler machen kann(in unserem Beispiel nur 2). Dann wird nach jedem dieser Moves die Funktion rekursiv aufgerufen mit der neuen Position(also dem Brett nach dem Zug von Weiß/Schwarz) der Tiefe -1 und dem anderen Spieler(indem fall False oder True). Die Funktion geht solange, bis man jede Position durchgerechnet hat und gibt bei jedem Aufruf den kleinsten/größten(größten Wert für Weiß, kleinsten Wert für Schwarz) Wert der Kinder zurück.

Optimierung des Algorithmus mit Alpha Beta-Pruning

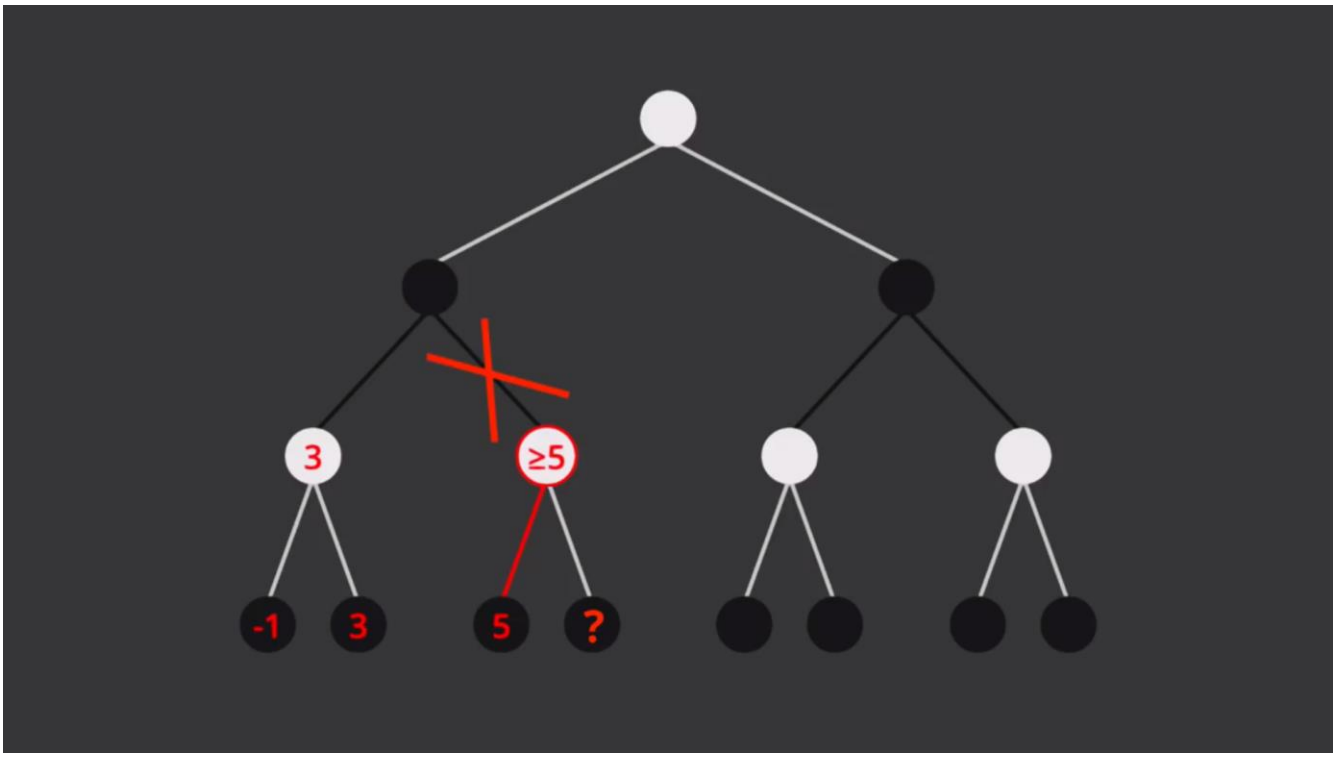
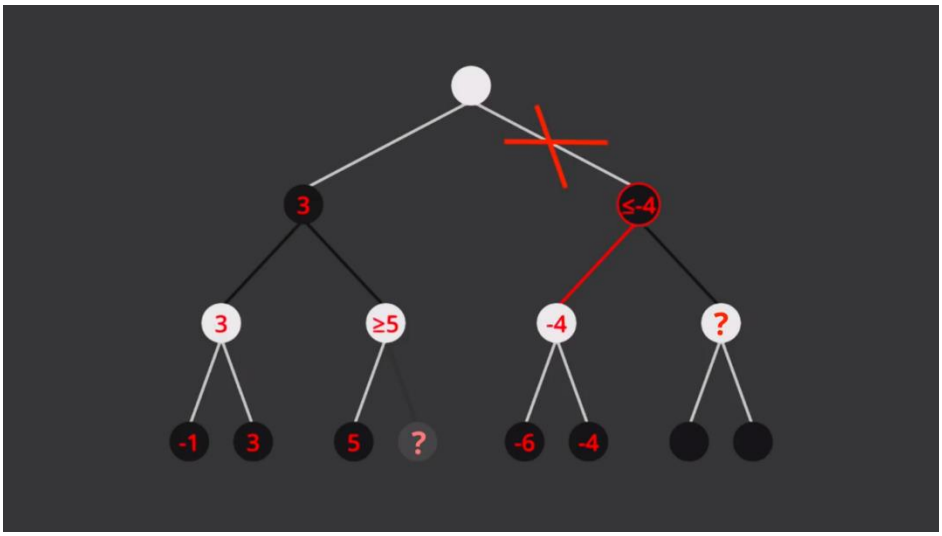


Bild 8 Alpha Beta-Pruning

Siehe Bild 8 Alpha Beta-Pruning: Anfangs verläuft der Alg. in diesem Beispiel gleich dem Minimax, bis er auf das erste Kind des letzten Astes trifft. Der Computer weiß, dass der Linke Ast zu einer Position von +3 für Weiß führen würde und nun sieht er das der Rechte Ast, Weiß einen Zug geben würde, welcher schon +5 hat. +5 ist größer als +3 weshalb Schwarz niemals diese Position wählen würde, weil Schwarz eben schon eine bessere Option hat. Deshalb braucht man jetzt keine Zeit mehr mit dem Ast mehr verschwenden und kann ihn also „abschneiden“ und so tun als gäbe es ihn nicht.



Das Gleiche passiert jetzt auf der rechten Seite des Baumes(Bild 9 Alpha Beta-Pruning 2). Der Computer sieht das mindestens -4 oder niedriger herauskommen würde, da Weiß schon eine bessere Option hat(die +3 auf der linken Seite), braucht es den Rest auf der rechten

Hälfte des Baumes nicht mehr berechnen.

Bild 9 Alpha Beta-Pruning 2

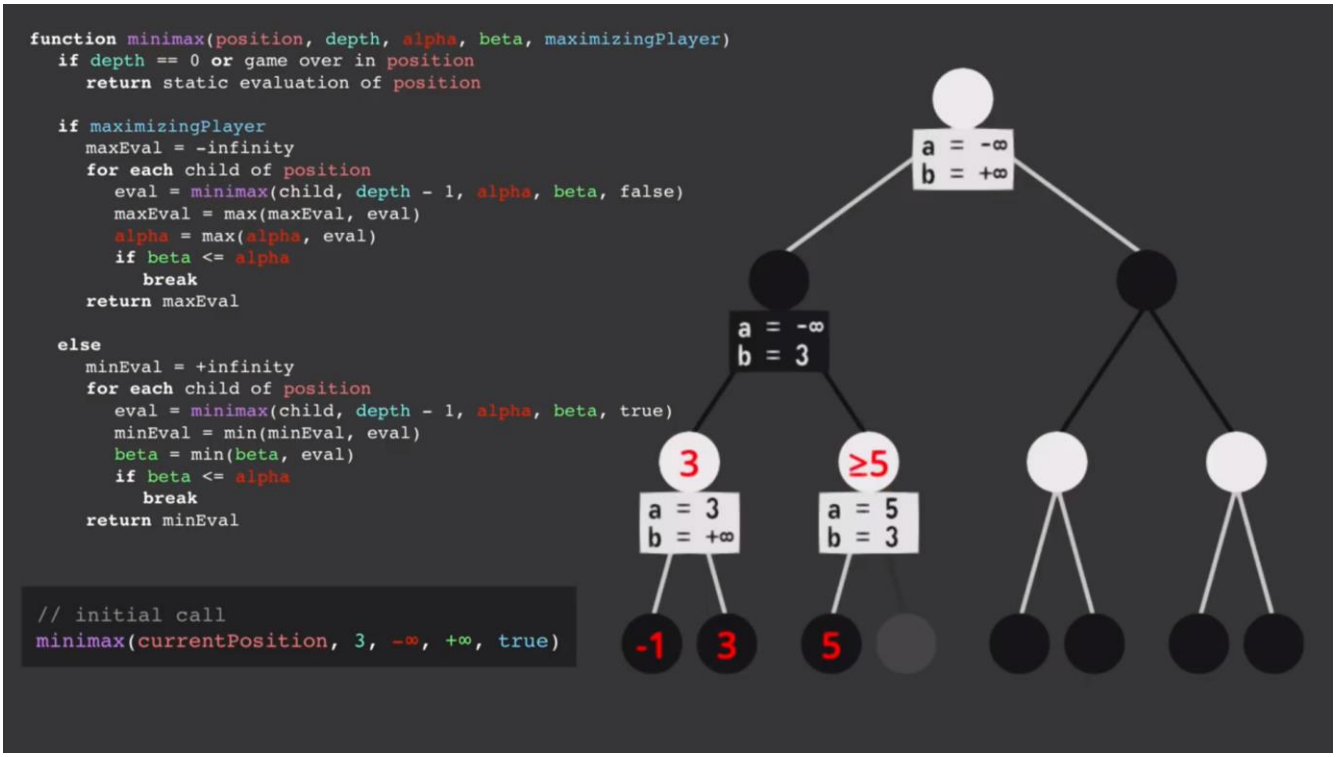
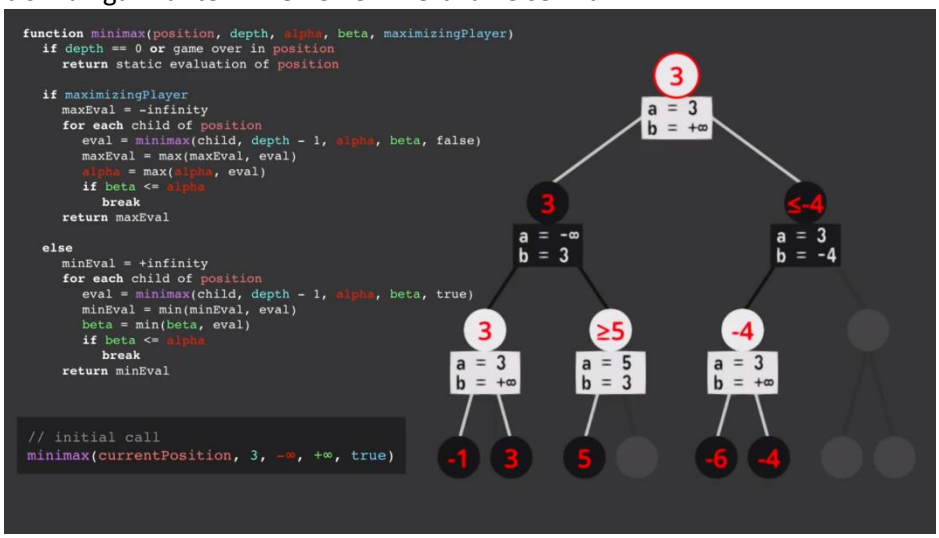


Bild 10 Alpha Beta-Pruning Pseudocode

Wie man auf Bild 10 erkennen kann, gibt es keinen großen Unterschied zum minimax im Pseudocode, die Funktion bekommt 2 zusätzliche Parameter „alpha“ und „beta“ welche beim ersten Aufruf auf –infinity und +infinty gesetzt werden(damit am Anfang nicht ausversehen ein Ast abgeschnitten wird). Im Verlauf des Programms wird Alpha dann immer auf den max. Wert und beta auf den min. Wert gesetzt im Zusammenhang mit dem letzen alpha/beta Wert und dem Wert des Kindes. Danach wird geschaut ob der beta Wert größer oder gleich dem alpha wert ist. Wenn ja kann das Programm bei diesem Ast aufhören weiter zu berechnen.

Auf Bild 10 kann erkennt man wie Anfangs die Werte –infinty und +infinty weiter nach unten gegeben werden bis man ganz unten ihnen einen Wert zuweisen kann.



Auf Bild 11 sieht man noch den Baum mit vollständigen abschneiden der unnötigen Äste.

Bild 11 [Alpha Beta-Pruning vollständiger Baum]

Alpha Beta-Pruning funktioniert am besten, wenn man „interessante“ und gute Moves als erstes probiert, da man sonst dem Glück ausgesetzt ist. Man sollte also die Moves sortieren, eine Idee wäre es Moves bei denen man eine Figur schlägt nach vorne zu setzen (oder auch: wenn ein Bauer am Ende vom Brett ist und sich verwandeln kann etc.).

Alpha Beta Pruning(so wie Minimax) wird umso besser desto mehr Zeit er hat, bzw. wie Tief er nach Moves schaut, deshalb könnte man auch einfach das Level der K.I anpassen indem man ihr weniger Zeit gibt oder eine geringere Tiefe.

Der eigentliche schwierige Teil bei der K.I wird sein die Methode, welche das Brett auswertet zu schreiben. Dort muss man sehr viel beachten und es ist von Situation zu Situation verschieden, z.B.: muss man nicht nur auf die Anzahl seiner Figuren im Zusammenhang zu den Figuren des Gegners schauen, es muss auch auf die Position der Figuren (ein Springer am Rand ist nicht so gut wie einer in der Mitte) geachtet werden. Aber wie gesagt das ist alles sehr Situationsabhängig, viele Grandmaster opfern sogar Figuren nur um ein paar eigene Figuren in eine bessere Position zu bringen.

Auch besonders wichtig ist es, das Endgame gut hinzubekommen, das ist Knifflig da man beim Endgame auch seinen König „aktivieren“ (Einsetzen um Gegnerische Bauern zu blocken und den eignen Bauern helfen durchzubrechen) muss. Die K.I muss außerdem Basic Check Mates mit Figuren wie z.B.: zwei Türmen (Ladder Check Mate) perfekt draufhaben, weil ein Fehler im Endgame oft das Spiel kosten kann

3.2.5. Ablauf eines Spiels

Zuerst muss man klären was ein Spieler, vor und während des Spieles machen kann, mit unserem UseCase Diagramm(Siehe Datei UseCaseDiagramm.mdj) versuchen wir genau das zu beschreiben.

Vor dem Start des Spiels kann das Design der Figuren und des Bretts geändert werden(Select board Style, select Piece Style). Sowie soll Spieler1 entscheiden können ob er gegen eine K.I spielen möchte oder lokal gegen einen Menschen(select Player Type). Sobald Spieler1 dann auf „Play“ drückt, beginnt das Spiel und beide Spieler können dann Züge machen(Natürlich nacheinander und Weiß beginnt). Beide Spieler können zu jedem Zeitpunkt nachdem Play gedrückt wurde, aufgeben und können nachdem ein Zug gemacht wurde auch den Zug

wieder zurücknehmen(Falls man sich verklickt hat). Die K.I wird logischerweise niemals aufgeben oder einen Zug zurücknehmen können. Diese beiden Optionen richten sich mehr an menschliche Spieler.

Auch haben wir ein Sequenz Diagramm(Siehe Datei SequenzDiagramm.mdj) erstellt, welches versucht zu erklären wie ein Zug abläuft, dabei geht es darauf ein, wie die Packages: Model, Controller und View miteinander kommunizieren und es soll nochmal deutlich machen, dass die View nie direkten Zugriff auf das Model haben sollte und umgekehrt.

3.2.6. Zeitplan/TODO

Unser Zeitplan(Siehe Datei Zeitplan.xslx)