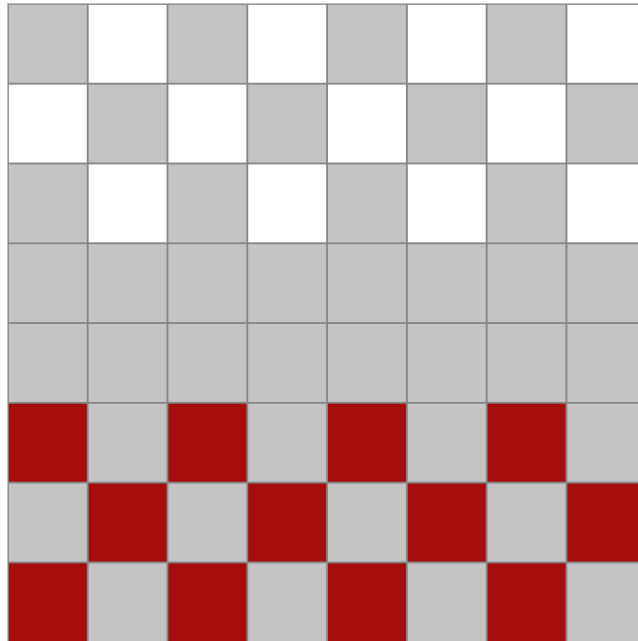


Programm zum Vergleich verschiedener Ansätze zur Lösung nicht effizient lösbarer Probleme anhand des Brettspiels Dame



Facharbeit für den Jugend-Forscht Wettbewerb

betreut von

Herr Frerkes

am

Steinhagener Gymnasium
Stufe Q2

14.01.2018

von

Marco Adamczyk

Till Brinkmann

Can Ward

Download des Programm und der Dokumentation:
<https://github.com/Gametypi/Checkers-Simulation>

Abstract

In unserer Facharbeit beschäftigen wir uns mit den verschiedenen Methoden einen Gegenspieler für das Brettspiel Dame zu programmieren. Wir haben grundsätzlich zwischen zwei unterschiedlichen Ansätzen differenziert. Zum einen gibt es die konventionellen Lösungsalgorithmen von Nullsummenspielen wie beispielsweise der MiniMax-Algorithmus. Diesen haben wir auf verschiedene Weisen implementiert. Einmal ohne Verbesserungen, mit Multithreading und mit Alpha-Beta Suche und diese daraufhin auch verglichen und bewertet.

Zum anderen haben wir uns Lösungen durch maschinelles Lernen angeschaut und haben ein evolutionsbasiertes Neuronales Netzwerk implementiert.

Um jedoch diese Ansätze auszuprobieren und letztendlich auch auswerten zu können, benötigen wir ein Programm, welches uns ermöglicht all dies zu testen. Deshalb haben wir ein Programm mit einer Benutzeroberfläche entwickelt, mit welchem wir die verschiedenen Algorithmen gegeneinander spielen lassen können. Besonders haben wir darauf geachtet, dass es modular bleibt und man jederzeit Algorithmen hinzufügen und entfernen kann, um auch anderen Personen, die daran interessiert sind, zu ermöglichen eigene Algorithmen zu schreiben und diese zum Programm hinzuzufügen ohne dieses zu verändern. Das ist auch einer der Gründe warum wir den Quellcode als auch die Dokumentation vollständig auf Englisch geschrieben haben.

Der Vergleich dieser Ansätze fand unter verschiedenen Aspekten statt, damit die Vor- und Nachteile vollständig dargestellt werden können.

Deshalb müssen die Ergebnisse auch häufig differenziert betrachtet werden, denn es kann nicht gesagt werden, dass ein Ansatz allgemein besser ist, sondern nur unter einzelnen Gesichtspunkten mehr Vorteile besitzt. Was besser ist entscheidet sich dadurch, welchen Nutzen dieses Programm haben soll und was dem Nutzer besonders wichtig ist. Diese Ergebnisse können auch abstrahiert betrachtet werden und lassen auf die grundlegenden Unterschiede zwischen Algorithmen und Neuronalen Netzen in verschiedenen Kontexten schließen.

Inhaltsverzeichnis

1.	Einleitung	2
2.	Checkers Simulation 2.0	3
2.1	Warum Dame?	3
2.2	Zielsetzung/Anforderungen	4
2.3	Umsetzung/Implementation	
2.4	Verbesserungen/Zusammenfassung	5
3.	MinMax	6
3.1	Konzept des Algorithmus	6
3.2	Variationen	9
	3.2.1 Alpha-Beta Suche	
	3.2.2 Multithreading	
3.3	Effizienzvergleich	10
3.4	Zusammenfassung	
4.	Das Neuronale Netzwerk	11
4.1	Einführung	11
4.2	Umsetzung	12
4.3	Lernprozess/Optimierung	14
4.4	Zusammenfassung	
5.	Alternative Ansätze	15
5.1	Datenbank-basierter Spieler	15
5.2	Zufallsbasierter Spieler	
6.	Vergleich/Fazit	16
7.	Quellenverzeichnis	17
8.	Anhang	18

1. Einleitung

Schon immer war es eines der Ziele der Menschheit Problemstellungen so effizient wie möglich zu lösen. Denn gerade durch die Lösung vieler Probleme konnte sich die Menschheit im Lauf der Geschichte schnell weiterentwickeln. Zunächst mit den bloßen Verstand, danach mit immer komplexeren Hilfsmitteln. Mit Rechenmaschinen wie zum Beispiel der Pascaline¹ wurden bereits im Mittelalter die Grundlagen für das Lösen komplizierter Aufgaben gelegt. Es wurde schnell erkannt, dass es Aufgaben und Problemstellungen gibt, die das Gehirn nicht so schnell bewältigen kann oder es sich einfach nicht lohnt diese zu bewältigen, da der Mensch sich in der Zwischenzeit wichtigeren Dingen zuwenden kann.

In der heutigen Zeit hat sich neben den anderen Naturwissenschaften besonders die Informatik dieser Menschheitsaufgabe angenommen und ist daher immer auf der Suche nach besseren Möglichkeiten Probleme zu lösen.

Neue Computersysteme und besonders Supercomputer sind in der Lage zur Zeit bis zu $93,0 \text{ PFLOPS}^2$ zu berechnen und diese Rechenleistung steigt immer noch exponentiell (Moorsches Gesetz). Jedoch sind manche Probleme, trotz dieser Zunahme, mit konventionellen Algorithmen nicht lösbar, wie beispielsweise das Traveling-Salesman Problem³ eindeutig zeigt.

Doch gerade diese stark zunehmende Rechenleistung lässt nämlich zunehmend auch neue Lösungsmöglichkeiten zu, wie beispielsweise die des maschinellen Lernens. Deshalb ist es immer wichtig auf der Suche nach neuen Ansätzen zu sein und diese unter diversen Aspekten mit den bewährten Verfahren zu vergleichen. Doch welche Methoden der Problemlösung sind die effektivsten, genauesten, zeitsparendsten, passendsten und zukunftsichersten für den jeweiligen Zweck?

Genau dies wollen wir in dieser Facharbeit annähernd und ansatzweise am Beispiel des Brettspiels Dame erläutern. Durch diese Bedeutung und Aktualität des Themas haben wir uns entschieden selbständig den Vergleich zu schaffen zwischen konventionellen Algorithmen und einem Neuronalen Netzwerk, dass den Ansatz des maschinellen Lernens verfolgt.

Zunächst gehen wir auf das Problem ein, dass wir mithilfe dieser Methoden lösen wollen.

Nach dieser theoretischen Einführung wollen wir auf die Umsetzung dieses Projektes eingehen, indem wir zunächst das Augenmerk auf die Grundlage legen, die uns den Vergleich erst ermöglicht hat. Es ist neben den Lösungsansätzen auch ein essenzieller Bestandteil dieses Projekts, der uns viel Zeit in der Entwicklung gekostet hat.

Im nächsten Abschnitt fokussieren wir uns auf die einzelnen Lösungsansätze und versuchen diese in der Tiefe auszuarbeiten und zu erklären. Besonders haben wir versucht unsere Implementierung der jeweiligen Methode zu präsentieren und zu erklären. Außerdem gehen wir noch auf andere Methoden der Problemlösung ein, die wir aber aus diversen Gründen nicht in unser Programm aufgenommen haben, um dem Leser einen besseren Überblick zu verschaffen.

Als letzten Schritt vergleichen wir diese zuvor erklärten Wege der Problemlösung an unserem Beispiel und geben ein differenziertes Fazit wieder, indem wir sie unter unterschiedlichen Aspekten vergleichen. Des Weiteren versuchen wir grundlegend in jedem größeren Abschnitt dieser Arbeit unsere Herangehensweise zu reflektieren.

1 Eine um 1645 entwickelte Rechenmaschine vom französischen Mathematiker Blaise Pascal. Sie basiert auf einem mechanischen Konstrukt aus Zahnrädern und Sperrklinken. Sie wurde ursprünglich dafür entwickelt Pascals Vater die Arbeit als Steuerbeamter zu erleichtern.

2 Steht für $93 \cdot 10^{15}$ Floating Point Operations Per Second. So schnell rechnet der chinesische Supercomputer Sunway-TaihuLight.

3 Ein Problem der Mathematik und Informatik, dass nicht effizient lösbar ist, also einen annähernd exponentiellen Aufwand aufweist.

2. Checkers Simulation 2.0

2.1 Warum Dame?

In diesem Teil wollen wir erklären warum wir das Brettspiel Dame als Grundlage für unser Projekt gewählt haben. Zunächst wollten wir die Anforderungen an unser Problem definieren:

- Es war uns wichtig, dass das Problem weit verbreitet ist und viele Leute es kennen. Andere Personen können sich somit einfacher in unser Projekt rein arbeiten.
- Ein weiterer nicht zu vernachlässigender Aspekt ist der Programmieraufwand.
- Zudem sollte es vollständig berechenbar sein, da so garantiert wird, dass es theoretisch sowohl vom Algorithmus, als auch neuronalen Netzwerk gelöst werden kann.
- Es sollte nicht effizient lösbar sein, also einen nahezu exponentiellen Aufwand besitzen.

Besonders das Brettspiel Dame erfüllt diese Voraussetzungen, denn es ist ein weltweit populäres Spiel, welches mit ähnlichen Regeln auf der ganzen Welt gespielt wird. Dadurch ist es auch wahrscheinlich, dass den Lesern das Verstehen der Facharbeit leichter fällt. Aber auch wenn die Regeln nicht beherrscht werden, ist das Spiel leicht zu erlernen, da es nur eine geringe Anzahl Spielmechaniken gibt. [Regeln im Anhang]

Des Weiteren lässt sich das Spiel durch die geringe Komplexität Regeln und Mechaniken mit einem relativ geringen Aufwand implementieren.

Es ist ein Zwei-Personen-Nullsummenspiel, da es entweder Sieg, Niederlage oder Unentschieden geben kann.

Außerdem gibt bei diesem Spiel keinen Zufall, es ist ein Spiel mit perfekter Information⁴ und ist nicht unendlich lang, da nach einer gewissen Anzahl von Zügen ein Unentschieden erzwungen wird.

Somit besitzt es ein eindeutiges Spielergebnis, wodurch sich vermuten lässt, dass das Spiel auch berechenbar ist. Dies wurde auch durch den Algorithmus Chinook bewiesen, welcher nach mehreren Jahren das Spiel komplett berechnete. Daher muss es auch eine endliche Anzahl an Spielsituationen geben. Diese kann durch Kombinatorik berechnet werden:

$$\sum_{s_{normal}=1}^{12} \sum_{s_{dame}=0}^{s_{normal}} \sum_{w_{normal}=1}^{12} \sum_{w_{dame}=0}^{w_{normal}} \frac{32!}{(s_{normal}-s_{dame})! \cdot s_{dame}! \cdot (w_{normal}-w_{dame})! \cdot w_{dame}! \cdot (32-(w_{normal}+s_{normal}))!}$$

Man kann hier das Prinzip der Berechnung für Permutation mit Wiederholung anwenden. Die Anzahl der besetzbaren Felder bei einem standard Dame 8*8 Feld liegt bei 32, somit gibt es auch 32 Möglichkeiten die Objekte auf dem Spielfeld anzuordnen. Man muss dabei zwischen den unterschiedlichen Figuren Typen, die in unserer Formel durch die jeweilige Variable repräsentiert werden, und den leeren Felder unterscheiden. Letzteres lässt sich aus der Differenz der besetzbaren Felder und der Anzahl der Figuren bestimmen.

Des Weiteren muss berücksichtigt werden, dass diese Berechnung mit jeder Kombinationen der Anzahl der normalen Figuren und Damen berechnet werden muss, da sich durch die Veränderung immer neue Anordnungen ergeben. Dies wurde durch die Summenzeichen umgesetzt.

Das Ergebnis dieser Rechnung liegt bei 6.011e+20.

Insgesamt wird hier deutlich, dass dieses Spiel trotz der relativ simplen Mechanik durch die hohe Anzahl an möglichen Spielfeldsituationen einen sehr hohen Berechnungsaufwand aufweist und sich somit die Frage stellt, ob neuronale Netze bei Nullsummenspielen eine Alternative zu konventionelle Algorithmen darstellen können.

⁴ Dies bedeutet, dass jeder Spieler zu jeder Zeit das Wissen über das passierte Spielgeschehen hat.

2.2 Zielsetzung/Anforderungen (Was soll es können):

Unser Programm, welches wir als Grundlage für den Vergleich genutzt haben, wurde von uns Checkers Simulation 2.0 genannt.

Es ist eine mit Java programmierte Anwendung, die bestimmte Dinge erfüllen sollte, damit wir die von uns programmierten Algorithmen darauf testen konnten.

Es besitzt eine grafische Oberfläche, mit einem Spielfeld, einer Eingabe und Ausgabekonzole und weiteren Optionen.

Es ist möglich Spiele sowohl gegen Computergegner, die dem Programm zu Verfügung stehen, als auch gegen einen anderen menschlichen Spieler zu spielen. Dies und andere Optionen kann man dann beim Start eines neuen Spiels auswählen. Man kann auch noch beispielsweise die Anzahl der Runden bestimmen, die gespielt werden oder ob das Spiel ausgewertet werden soll. Die Auswertung speichert alle wichtigen Werte in Dateien, sodass es im Nachhinein ausgewertet werden kann.

Auf der eigenen Konsole werden während und nach dem Spiel alle wichtigen Ereignisse, Fehler und Warnungen ausgegeben. Durch die Eingabe kann man auch Zugriff auf die Algorithmen kriegen.

Zu dem gibt es die Möglichkeit ein anpassbares neuronales Netz zu trainieren. [Bedienungsanleitung im Anhang]

Das Programm wurde von uns mit der Programmiersprache Java entwickelt, da uns eine objektorientierte Programmierung wichtig war. Diese hat es uns ermöglicht das Programm sinnvoll zu strukturieren und damit zugänglicher für andere zu machen. Es ist auch einfacher Teile des Quellcodes wiederzuverwenden und Fehler zu beheben. Zudem gibt es noch weitere Vorteile, die unsere Entscheidung maßgeblich beeinflusst haben. Java ist eine weit verbreitete Programmiersprache, die besonders auf Portabilität und Parallelisierbarkeit ausgelegt ist.

Natürlich müssen auch die Schwächen kurz erwähnt werden, denn Java ist in vielen Bereichen nicht so effizient wie andere Programmiersprachen. Jedoch, unter Berücksichtigung der anderen Aspekte, lässt sich sagen, dass Java für unsere Ansprüche eine durchaus berechtigte Wahl gewesen ist.

Die Entwicklungsumgebung, die wir gewählt haben, ist Eclipse. Diese IDE hat sich als sehr zuverlässig erwiesen und bietet einige Vorteile, wie zum Beispiel eine größere Anzahl an Funktionen, die das Programmieren vereinfachen und die besseren Debugmöglichkeiten.

Außerdem haben wir als Sprache für den Quellcode und die Dokumentation Englisch verwendet. Englisch ist eine weltweit verbreitete Sprache, die mittlerweile in der Arbeitswelt aufgrund der fortschreitenden Globalisierung als der Grundlage der Kommunikation gilt. Unser Programm wird dadurch nicht nur auf den deutschsprachigen Raum begrenzt, sondern für jeden zugänglich sein, der sich damit befassen möchte.

2.3 Umsetzung/Implementation

Die diversen Klassen des Programms haben wir in verschiedene Pakete zusammengefasst um sie inhaltlich voneinander abzugrenzen.

Zum einen gibt es das Paket checkers, welches alle Klassen zusammenfasst die direkt etwas mit dem Spiel und dessen Ablauf zu tun haben. Das wäre die Figure, sie repräsentiert eine Damefigur mit Positions-, Typ-, und Farbvariablen⁵.

Diese Figuren werden in einer Klasse namens Playfield zusammengefasst. Sie bietet mehrere Methoden um die Situation auf dem Spielfeld zu erfassen, zum Beispiel isOccupied(x, y) oder getFigureQuantity(FigureColor), aber auch zu verändern wie executeMove(Move).

Jeder Zug in einem Spiel wird auch als eigenes Objekt vom Typ Move behandelt. Move enthält alle nötigen Variablen um einen Zug eindeutig zu identifizieren und einige statische Methoden die eine Spielfeldsituation analysieren und eine Liste aller möglichen Züge, nur Schlagzüge oder nur Schritte zurückgeben.

Als vorletztes findet sich die GameLogic. Sie koordiniert ein gesamtes Spiel das auch aus mehreren Einzelrunden bestehen kann. Dafür erhält sie in ihrer startGame() Methode die zwei Spieler, den (optionalen) Spielnamen, die Anzahl der Runden, die Geschwindigkeit für nicht menschliche Spieler, einen Wahrheitswert, der aussagt ob das Spiel angezeigt werden soll⁶ und noch einen der anzeigt ob man auf dem gerade angezeigten Spielfeld weiterspielen oder ein neues nutzen möchte. Eine weitere Aufgabe der GameLogic ist die Zugvalidation mit der Methode testMove(Move, Playfield).

Einer der kürzesten, aber definitiv wichtigsten Klassen des Projekts ist das Interface Player.

⁵ Durch die Farbe kann erkannt werden, zu welchen Spieler die Figur gehört.

⁶ Nützlich wenn man statistische Ergebnisse aus dem Spiel von zwei nicht menschlichen Spielern bekommen möchte. Das Spiel läuft ohne Anzeige nochmal ein bisschen schneller.

public interface Player
<pre> public: void prepare() void requestMove() String getName() boolean acceptDraw() void saveInformation() </pre>

Abbildung 2.1: Methoden des Player Interfaces

Dieses Interface muss von jedem Algorithmus und jeder sonstigen Klasse die ein Spiel in unserem Programm spielen möchte implementiert werden. Alle nötigen Informationen können nur über dieses Interface kommuniziert werden. Dadurch erreichen wir eine größtmögliche Modularität und können Spieler beim Start des Programms dynamisch laden.

Das nächste Paket heißt evaluation. Es ist eine Sammlung zweier Klassen die genutzt werden um ein Spiel zu dokumentieren und eine Vielzahl von Informationen zu speichern, die hilfreich sind um Spielstärke und Schnelligkeit der einzelnen Spieler zu bewerten. Zusätzlich erhält jeder Spieler durch die `saveInformation()` Methode nach jeder Runde die Möglichkeit eigene Informationen zu speichern.

Das Paket gui enthält alle Klassen die etwas mit der graphischen Oberfläche des Programms zu tun haben.

Am wichtigsten ist hier GUI. Das ist sowohl die Hauptklasse unseres Programms als auch das Hauptfenster.

Im Konstruktor werden alle graphischen Komponenten initialisiert die das Projekt benötigt. Die GUI hat Zugriff auf die meisten Klassen, weil sie auch Einstellungen weitergibt. Die weiteren Klassen sind größtenteils Panele oder Fenster die bestimmte Einstellungen oder in sich geschlossene graphische Komponenten enthalten. Das Playfieldpanel implementiert zusätzlich noch das Interface Player um dem Nutzer das Spielen zu ermöglichen.

Als Nächstes kommt das Paket network. Es enthält die Klassen, die alle Netzwerkaktivitäten des Programms koordinieren.

Zuletzt gibt es noch zwei Pakete, datastructs und utilities, die Klassen und Methoden enthalten die an vielen Stellen genutzt werden und deswegen als eigenes Paket in das Projekt eingebracht wurden.

2.4 Verbesserungen/Zusammenfassung

Insgesamt ist dieses Programm ein vielseitiges und starkes Tool zum Auswerten von Algorithmen. Jes bietet einige Funktionen und lässt sich relativ intuitiv bedienen. Eine der Schwächen ist, dass die Struktur manchmal nicht optimal ist und das es dadurch etwas unübersichtlicher ist. Um dem entgegenzuwirken, könnte man einige Dinge noch in unterschiedliche Methoden und Klassen auslagern. Außerdem haben wir manche Funktionen, die wir uns zu Beginn vorgenommen haben, nicht implementieren können.

3. MiniMax Algorithmus

3.1 Konzept

Der MiniMax Algorithmus⁷ ist ein informatischer Ansatz zur Lösung von Zwei-Spieler-Nullsummenspielen. Er basiert auf dem MiniMax Prinzip in dem zur Voraussage eines Gewinns immer der schlechtest mögliche Weg genommen wird (daher auch pessimistisches Prinzip). Dadurch lässt sich immer ein absolut sicher erreichbares Ergebnis voraussagen.

In unserem Fall bedeutet das also, dass davon ausgegangen wird das der Gegner immer den bestmöglichen Zug macht (das wäre der schlechteste Fall).

Es muss also versucht werden, mit dem gewählten Zug die maximale Zugqualität des Gegners zu minimieren und die minimale eigene Qualität zu maximieren (daher auch der Name). Die eben genannte Qualität eines Zuges wird im Algorithmus durch eine Bewertungsfunktion errechnet. Diese Bewertungsfunktion gibt hohe Werte für Züge die gut für den Spieler sind und niedrige, meist negative Werte für Züge von denen der Gegner profitiert. Eine optimale Bewertungsfunktion würde also den Wert 1 für das Gewinnen des Spiels und -1 für das Verlieren annehmen. Ein Gleichstand würde mit 0 bewertet werden.

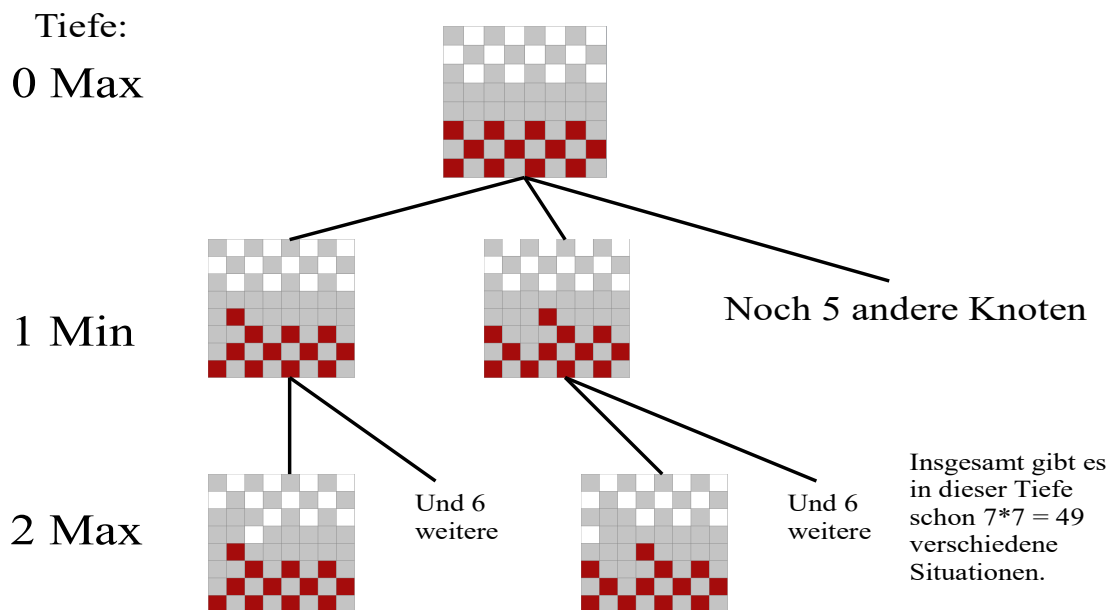


Abbildung 3.1: Modell eines MiniMax Suchbaums mit Tiefe 3 ausgehend von der Dame Startposition

In obenstehender Abbildung sehen wir die Arbeitsweise des MiniMax mit einer festen Tiefe. In Tiefe 0 wird für jeden Zug, der auf dem momentanen Spielfeld möglich ist, ein neuer Ast erstellt. Jeder dieser Äste erstellt neue Zweige nach dem gleichen Prinzip, allerdings wird hier die ursprüngliche Spielfeldsituation mit dem ausgeführten Zug des Elternknotens als Ursprung genommen. Der aufgebaute Suchbaum wird post-order⁸ traversiert.

Die Evaluation wird nur in den Blättern⁹ durchgeführt, das Ergebnis wird dabei zum Elternknoten hochpropagiert. Dieser vergleicht alle Ergebnisse der Kindknoten und bestimmt, je nachdem ob es eine Min-/ oder Max-Ebene ist den größten oder kleinsten Wert. Dieser Wert wird wiederum an die Elternknoten weitergegeben bis alle Werte bei der Wurzel angelangt sind. Die Wurzel führt dann die Aktion aus die mit dem Ast assoziiert wird, der den besten Wert zurückgegeben hat.

Das ermöglicht dem MiniMax je nach Suchtiefe mehrere Züge vorauszuberechnen und so einen besseren Überblick über Langzeitfolgen der Aktion zu bekommen.

⁷ Ein Algorithmus kann als eine Handlungsvorschrift beschrieben werden, welche durch eine systematische und definierbare Vorgehensweise eine Problemstellung lösen kann.

⁸ Die post-order Traversierung beschreibt in der Informatik eine Methode der Suchbaumtraversierung. Hierbei wird immer zuerst der linke Kindknoten, danach die rechten Kindknoten und zum Schluss die Wurzel selbst abgearbeitet.

⁹ Ein Knoten, der keine Kindknoten besitzt wird als Blatt bezeichnet.

Wie in Abbildung 3.1 zu sehen ist, wird die Anzahl der zu berechnenden Situationen pro Tiefe exponentiell größer. Linear dazu steigt auch der Rechenaufwand und somit auch die Rechenzeit. Dieses lässt sich aus den folgenden, empirisch mit der Auswertungsfunktion der Checkers Simulation 2.0 erhobenen Ergebnissen schlussfolgern. Dafür wurde der Durchschnittswert aus 100 Spielen ermittelt.

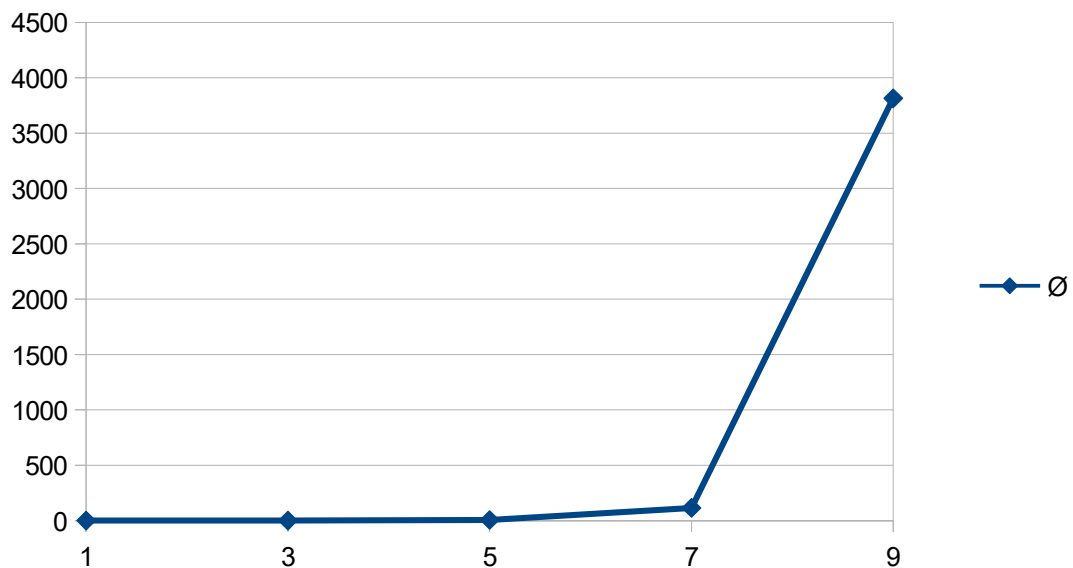


Abbildung 3.2: Durchschnittliche Zugzeiten des MiniMax in Millisekunden mit verschiedenen Maximaltiefen

Ein durchschnittliches ausgewogenes Damespiel besteht aus etwa 50 bis 70 Zügen. Um eine sichere Gewinnchance zu berechnen müsste man diesen Suchbaum im schlimmsten Fall also bis zu einer Tiefe von 70 generieren. Es ist also aufwendig den Baum bis zu einer Situation aufzubauen in der sich eindeutig bestimmen lässt, ob das Spiel gewonnen wird oder nicht. Bei den meisten Problemen kann eine ideale Bewertungsfunktion also nicht verwendet werden da der Zeit-/ oder Rechenaufwand einfach zu hoch wäre.

Daher haben wir eine Bewertungsfunktion entwickelt, die andere Parameter zur Bewertung nutzt:

$$Wert_{Blatt} = (Figurenanzahl_{Tiefe\ 0; \text{Gegner}} - Figurenanzahl_{Tiefe\ Max; \text{Gegner}}) + \square (Figurenanzahl_{Tiefe\ Max; \text{Spieler}} - Figurenanzahl_{Tiefe\ 0; \text{Spieler}})$$

Diese Funktion berechnet, wie viele Figuren des Gegners geschlagen wurden und wie viele Figuren von einem selbst. Je mehr Figuren des Gegners geschlagen wurden, desto höher ist die Bewertung.

Für jede eigene geschlagene Figur wird allerdings ein Punkt abgezogen. Dadurch lässt sich, ohne die Endsituation zu kennen, eine zuverlässige Gewinnprognose erstellen.

Unsere Implementation dieses Algorithmus besteht aus 3 Klassen. Die erste ist der MiniMaxPlayer selbst. Diese Klasse implementiert das Player Interface und dient damit hauptsächlich als die Schnittstelle zum restlichen Programm und als Wurzel des Suchbaums.

Die zweite Klasse ist der MiniMaxTask. Sie implementiert den Kern der Berechnung und repräsentiert einen Knoten des Suchbaums. Um nötige Informationen für Zugbewertung und Programmflusssteuerung, hauptsächlich die maximale Tiefe um das Ende des Baumes zu bestimmen und die Figurenanzahlen die in der Bewertungsformel genutzt werden, für jeden MiniMaxTask verfügbar zu machen, gibt es noch einen MiniMaxManager der diese Daten speichert.

MiniMaxPlayer	MiniMaxTask	MiniMaxManager
bestMove bestValue prepare() requestMove()	value move playfield compute() evaluateMove()	playerColor enemyColor maxDepth updateFigureCounts() getFigureQuantity()

Abbildung 3.3: Wichtigste Methoden und Variablen der MiniMax Klassen

Die wichtigsten Variablen im MiniMaxPlayer sind, neben allem was für den Player gebraucht wird wie Konsole und Gamelogic, der beste Move, welcher am Ende der Berechnung auf dem Spielfeld ausgeführt wird und die dazugehörige beste Bewertung. Fast alle Methoden sind vom Player Interface implementiert, die wichtigsten hier sind prepare(), welche einen für das startende Spiel passenden MiniMaxManager mit den richtigen Spielerfarben erstellt und requestMove(). Wenn diese Methode aufgerufen wird, startet eine neue MiniMax Berechnung. Dafür wird zuerst eine Liste aller möglichen Züge erstellt.

```
public void requestMove(){
    //start maximizing minMaxTask for every possible move
    List<Move> moves =
        Move.getPossibleMoves(manager.playerColor, gmlc.getPlayfield());
    [...]
```

Danach wird ein erster wichtiger Optimierungsschritt durchgeführt. Wenn nur ein Zug verfügbar ist, was oft passiert durch die Schlagzwang Regel[Regeln im Anhang], wird keine weitere Berechnung gestartet. Es gibt schließlich nur diesen einen Zug der ausgewählt werden kann.

```
    [...]
    if(moves.length == 1) {
        moves.toFirst();
        gmlc.makeMove(moves.get());
        return;
    }
    [...]
```

Wenn das nicht der Fall ist wird als nächstes der MiniMax Suchbaum aufgebaut und der Wert eines jeden Kindknotens evaluiert. Wenn der Wert besser ist als der Vorherige wird der momentane Wert der neue höchste Wert und der Zug der von diesem Knoten bewertet wurde der neue beste Zug. Am Ende der Methode wird dann der beste Zug ausgeführt.

```
    [...]
    float v;
    for(moves.toFirst(); moves.hasNext(); moves.next()){
        v = new MiniMaxTask(
            tmpman,
            moves.get(),
            gmlc.getPlayfield().copy(),
            1,
            //the next task is always not maximizing
            false
        ).compute();
        if(v > bestValue){
            bestValue = v;
            bestMove = moves.get();
        }
    }
    gmlc.makeMove(bestMove);
}
```

Der eigentliche MiniMax Such-/ und Auswahlprozess findet in der compute() Methode statt. Dort wird zunächst der zu bewertende Zug auf dem Spielfeld ausgeführt.

```
public float compute() {
    pf.executeMove(move);
    [...]
```

Daraufhin wird getestet, ob die maximale Tiefe erreicht ist. Wenn dies der Fall ist, wurde das Ende des Suchbaums erreicht und es kann das Ergebnis der Bewertung zurückgegeben werden.

```
[...]
if(depth >= manager.maxDepth){
    return evaluateMove();
}
[...]
```

Im MinMaxTask unterscheidet sich die Auswahl der Bewertungen insofern, dass hier nicht immer der beste Wert genommen wird sondern, wenn der Task nicht maximiert (also isMaximizing falsch ist), die minimalste Bewertung ausgewählt.

```
[...]
if(isMaximizing ? v > value : v < value){
    value = v;
}
[...]
```

Die oben genannte Bewertungsfunktion ist in der Methode evaluateMove() implementiert. Hier wird der Manager genutzt, um die ursprünglichen Figurenanzahlen zu erhalten.

```
private float evaluateMove() {
    return (manager.getFigureQuantity(manager.enemyColor)-
        pf.getFigureQuantity(manager.enemyColor))+
        (pf.getFigureQuantity(manager.playerColor)-
        manager.getFigureQuantity(manager.playerColor));
}
```

3.2 Variationen

3.2.1 Alpha-Beta Suche

Die Alpha-Beta Suche ist eine effizientere Alternative zur normalen kompletten Durchsuchung des aufgebauten Baums. Mit dieser Art der Suche können manche Teile des Baums ignoriert werden, da die sowieso nicht an den minimale Wert(alpha) und maximalen Wert(beta) gelangen können und somit keine Bedeutung mehr haben.

Diesen Vorgang nennt man Alpha oder Beta Schnitt. Die Anzahl der zu bewertenden Situationen sinkt also je mehr Schnitte gemacht werden. Insgesamt man der Algorithmus durchschnittlich 8421 Betaschnitte und 3101 Alphaschnitte pro Spiel. Dadurch kann die Rechenzeit drastisch verbessert werden.

Wie in unsere Implementierung für einen beta-Schnitt zu sehen ist wir hier ein ganzer Zweig nicht mehr berücksichtigt, weil unter bestimmten Bedingungen schon returnt wird. Die geschieht wenn der aktuelle Wert größer ist als der Wert von beta, welcher aus einer geringeren Tiefe übergeben wurde, da so

```
[...]
if(isMaximizing && v > value){
    value = v;
    if(v >= beta) {
        return value;
    }
    alpha = v;
}
[...]
```

3.2.2 Multithreading

Die Multithreading-Version des MiniMax gleicht in der grundlegenden Arbeitsweise dem „normalen“ MiniMax. Es wurde nur eine für Multithreading optimierte Implementierung genutzt. Hier wird jeder MiniMaxTask als eigene Aufgabe an einen Ausführungsservice abgegeben und parallel ausgeführt. Am Ende seiner Ausführung ruft jeder Task eine Callback Funktion seines Elternknotens auf mit der er das Ergebnis übergibt. Als Ausführungsservice wird ein von uns selbst programmierter StackExecutor verwendet, da dieser effizienter funktioniert als die Javabibliothek ForkJoinPool.

3.3 Effizienzvergleich

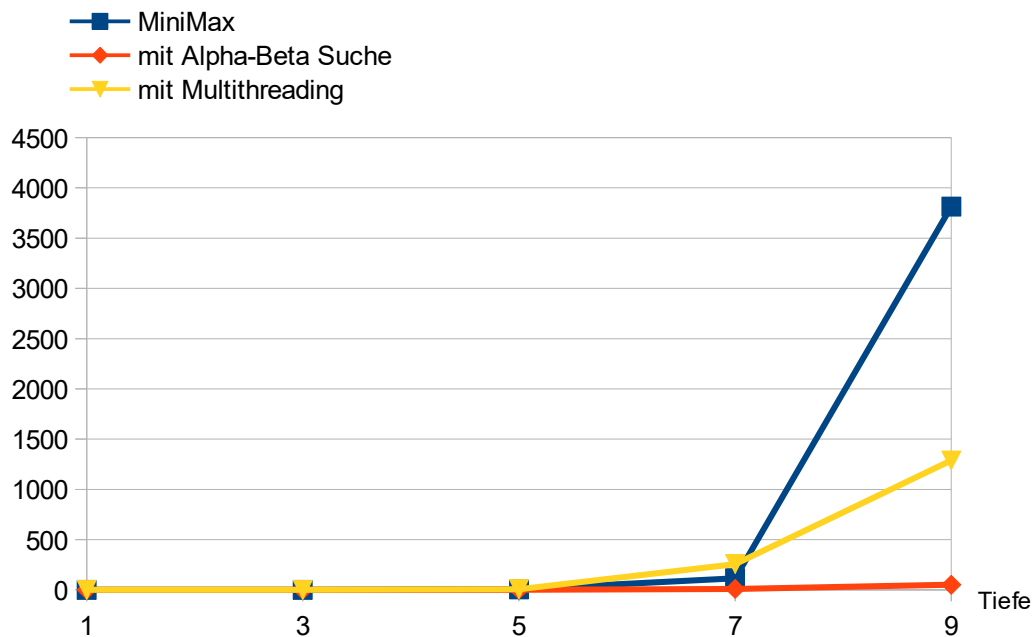


Abbildung 3.4: Durchschnittliche Zugzeiten aller MiniMax Varianten in verschiedenen Tiefen in Millisekunden.

Im direkten Vergleich zeigt sich, dass die Alpha-Beta Suche eine enorme Beschleunigung ermöglicht. Bei Tiefe 9 ist der MiniMax mit Alpha-Beta Suche etwa 75 mal so schnell wie der ohne Modifizierung und immer noch 25 mal so schnell wie die Variante mit Multithreading. Die Prozessorauslastung ist bei Alpha-Beta und „normalem“ MiniMax gleich. Sie liegt bei etwa $1 / \text{Anzahl der Prozessorthreads}$. Der Multithreading MiniMax verbraucht allerdings immer nahezu 100 Prozent der Prozessorleistung. Beim Verbrauch anderer Ressourcen wie Speicherplatz sind alle Varianten gleichauf. Jede Einzelne verbraucht als .jar Datei etwa 6 Kilobytes¹⁰.

3.4 Zusammenfassung

Zusammenfassend kann gesagt werden, dass der Mini-Max ein guter Lösungsweg ist, der die Berechnungen direkt im Spiel durchführt und sich somit als praktisch erweist. Zu dem kann er mit verschiedenen Varianten noch effizienter und schneller arbeiten. Am besten wäre es natürlich Alpha-Beta und Multithreading zu verbinden, jedoch birgt das einige Probleme, da die Alpha-Beta Suche sich nicht so einfach parallelisieren lässt. Die Implementierung nur 150 Zeilen lang und somit überschaubar. Man kann als Nachteil betrachten, dass die Berechnungsergebnisse nie gespeichert werden. Das ist zum einen nicht effizient und zum anderen ist es dadurch schwer ein ganze Spiel auf einer sehr hohen Spielstärke zu spielen (hohe Tiefe), da die Rechenleistung dafür einfach unglaublich hoch sein muss, wenn jeder Zug neu berechnet werden muss. Bei schwachen Cpu kann es auch durchaus sein, dass er schon bei relativ geringer Tiefe sehr lange für einzelne Züge braucht.

¹⁰ Von Windows auf einem NTFS Dateisystem berechnet.

4. Das Neuronale Netzwerk

4.1 Einführung

Der Mensch ist immer noch einer der besten Dame-Spieler in unserer Auswahl. Auch wenn er vielleicht nicht immer ein perfektes Spiel wie ein MiniMax mit hoher Tiefe spielt, kann er mit ein wenig Übung innerhalb von Sekunden relativ zuverlässig eine gute Entscheidung treffen und so selbst gegen den perfekt-spielenden Gegner noch einen Gleichstand erspielen. Diese Fähigkeiten sind in der Komplexität unseres Gehirns begründet. Es besteht aus Milliarden von Neuronen die alle parallel laufen, ununterbrochen unsere Sinneseindrücke verarbeiten und in Wissen oder konkrete Aktionen umwandeln.

Um diese Vielseitigkeit auch in der Informatik nutzen zu können, muss ein „Modell“ des Gehirns implementiert werden.

Mit anderen Worten ein künstliches neuronales Netz.

Ein neuronales Netzwerk beschreibt in der Informatik eine Struktur, die viele einzelne künstliche Neuronen mit teilweise unterschiedlichen Fähigkeiten in einer geschichteten Struktur verbindet in die „vorne“ eine Eingabe gegeben wird und die dann hinten eine Ausgabe basierend auf diesen Eingaben gibt. Die einzelnen Neuronen sind stark einer menschlichen Nervenzelle nachempfunden. Ein Neuron hat einen oder mehrere Inputs und einen Output (der allerdings zu mehreren anderen Neuronen führen kann). Meistens wird der Output berechnet indem alle Inputs summiert und dann durch eine Funktion verändert werden. Die Neuronen sind allerdings nicht direkt miteinander verbunden sondern kommunizieren über eine Verbindung. Diese Verbindung hat ein Gewicht, welches mit dem ankommenden Wert multipliziert und dann erst weitergegeben wird.

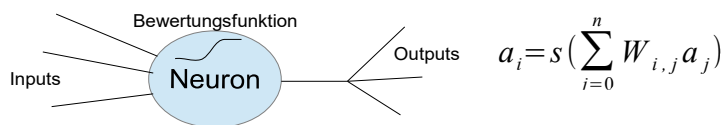


Abbildung 4.1: Modell eines Neurons und die Berechnungsfunktion für den Output

In dieser Formel wird der Outputwert als a von i , die Funktion als $s(x)$, die Gewichte der Verbindungen als W und die Ausgabe der Zellen der vorherigen Schicht als a von j beschrieben. Beispiele für die sogenannte Aktivierungsfunktion sind

die Sigmoidfunktion $s(x) = 1 / (1 + e^{-x})$,

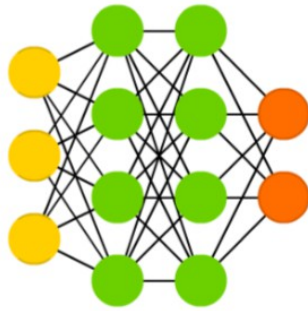
eine binäre Schwellwertfunktion mit Schwellwert t
$$\begin{aligned} x < t &\Rightarrow s_1 \\ x \geq t &\Rightarrow s_2 \end{aligned}$$
,

oder eine lineare Funktion $s(x) = m * x + b$.

In einem NN befinden sich also eine große Zahl von Parametern die verändert werden können. Dadurch wird ein hoher Grad an Universalität erreicht, da theoretisch jede Art von Information bei gleicher Struktur verarbeitet werden kann. Neuronale Netze sind also vereinfachte Modelle des Gehirns und sind deshalb sehr vielseitig einsetzbar und können viele unterschiedliche Aufgaben erledigen. Ein gutes Beispiel dafür, dass künstliche neuronale Netze auch komplexe Aufgaben wie das Spielen von Brettspielen erledigen können, ist das von Google DeepMind entwickelte AlphaGo.

4.2 Umsetzung

Deep Feed Forward (DFF) Bei unserem neuronalen Netzwerk haben wir uns für eine Deep Feed Forward Struktur entschieden, auf die später noch genauer eingegangen wird.



In nebenstehender Abbildung sind gut die vorher angesprochenen verschiedenen Arten von Neuronen zu sehen.

Das abgebildete neuronale Netz besteht aus Input-Neuronen, welche in der Abbildung gelb dargestellt werden, Hidden-Neuronen, welche in der Abbildung grün dargestellt werden, und Output-Neuronen, die orange dargestellt werden.

Abbildung 4.2: DFF Netzwerk

Ein Netz kann als deep bezeichnet werden wenn es mehr als eine Schicht von Hidden-Neuronen besitzt.

Ein Neuronales Netz bekommt immer Zahlen als Input. Bei einem Feed Forward-Netz berechnet man den Output indem erst die Input-Neuronen auf die Werte des Inputvektors gesetzt und dann die Layer und die Verbindungen nacheinander aktualisiert werden. Hierbei geben die Input-Neuronen den ihnen zugewiesenen Wert nur an die Verbindungen weiter. Die Verbindungen wiederum multiplizieren den Wert mit ihrem Gewicht und geben das entstandene Produkt dann an die Hiddenneuronen des ersten Hidden-Layers weiter. Wenn ein Hiddenneuron mehrere Werte von den Verbindungen bekommen hat und aktualisiert wird, addiert es diese Werte erst zu einer Summe, welche dann eine Aktivierungsfunktion, die auch Transferfunktion genannt wird, durchläuft. Wir benutzen für unser Projekt nur die Sigmoidfunktion¹¹. Nachdem alle Neuronen, eines Hidden-Layers, geupdatet wurden geben alle ihre berechneten Werte an die Verbindungen weiter, welche diese wieder mit einer Variable multiplizieren und weitergeben. Dieser Vorgang wird solange wiederholt bis die Werte schließlich an den Output-Layer weitergegeben werden. Die Output-Neuronen funktionieren wie die Hidden-Neuronen bis auf den Unterschied, dass sie den Wert nicht mehr weitergeben. Nachdem diese aktualisiert wurden, bilden ihre Werte den Output des Netzes. Das neuronale Netz besteht aus einer einzelnen Klasse. In ihr befinden sich die Variablen, die die Gewichte, der Verbindungen und den Bereich, indem die Sigmoidfunktion skaliert wird angeben. Die folgende Tabelle zeigt die Methoden der Klasse NN und beschreibt diese.

Methode	Beschreibung
<code>randomWeights()</code>	Allen Gewichten wird ein zufälliger Wert in einem festgelegten Bereich zugeordnet.
<code>vector_matrix_multiplication()</code>	Die Übergabe, Multiplikation mit den Gewichten und die Addition der Werte, der Neuronen, einer Schicht, haben wir hier durch eine Matrizenmultiplikation, bei der die Werte, der Neuronen einen Vektor und die Gewichte eine Matrix bilden, umgesetzt.
<code>sigmoid()</code>	Ein übergebener Wert wird in eine gestauchte und verschobene Sigmoidfunktion eingesetzt und das Ergebnis wird zurückgegeben.
<code>changeAll()</code>	Alle Gewichte werden mit zufälligen Werten, in einem festgelegten Bereich, multipliziert.
<code>changeFunction()</code>	Ein übergebener Wert wird in eine kubische Funktion, die für <code>changeAll()</code> gebraucht wird um die Zufallszahl, die für die Veränderung der Gewichte benötigt wird, zu skalieren, eingesetzt und das Ergebnis wird zurückgegeben.
<code>childFrom()</code>	Die Gewichte des neuronalen Netzes werden zu einer Mischung der Gewichte von zwei anderen Netzen geändert. Hierbei wird bei jedem Layer zufällig entschieden, von welchem Netz der Layer übernommen wird.
<code>run()</code>	Run nimmt als Parameter einen Inputvektor und gibt den Outputvektor zurück. Es wird also einfach das Netz

Abbildung 4.3: Methoden der Klasse NN

¹¹ Die Funktion wird allerdings, je nach Einstellung, skaliert und verschoben.

Um uns das speichern und Übergeben der Struktur und Variablen, die das neuronale Netz bei der Initialisierung benötigt, zu vereinfachen, haben wir die Klasse `NNSpecification` geschrieben, in der diese Werte gespeichert sind. Unsere neuronalen Netze besitzen 64 Inputneuronen, 640 Hiddenneuronen, welche gleichmäßig auf 10 Layer aufgeteilt sind und 64 Outputneuronen. Damit besitzen sie 40960 Verbindungen, mit jeweils einem Gewicht. Um ein neuronales Netz mit dem Brettspiel Dame zu verbinden haben wir die Klasse `NNPlayer` hinzugefügt. Diese hat erstens die Aufgabe das Feld der aktuellen Spielsituation an das neuronale Netz, das am Zug ist, weiterzugeben, wobei die ersten 32 Werte beschreiben auf welchen Feldern sich normale Figuren befinden und welche davon eine eigene und welche eine gegnerische ist. Hierbei steht eine 1 für eine eigene normale Figur, eine -1 für eine gegnerische normale Figur und eine 0 für ein leeres Feld oder eine Dame. Die zweiten 32 Werte beschreiben auf welchen Felder sich Damen befinden und welche davon eine eigene und welche eine gegnerische ist, hierbei steht eine 1 für eine eigene Dame, eine -1 für eine gegnerische Dame und eine 0 für ein leeres Feld oder eine normale Figur. Die zweite Aufgabe ist es basierend auf den Outputwerten einen Zug zu ermitteln, wobei von den ersten 32 Werten der höchste Wert das Feld angibt, zudem sich die Figur bewegen soll und von den zweiten 32 Werten der höchste Wert die Figur angibt, die sich dorthin bewegen soll, wobei nur Figuren berücksichtigt werden, für die dieser Zug möglich wäre. Am Ende wird die Entscheidung des Netzwerks bewertet. Für die Umsetzung des Trainings des neuronalen Netzes haben wir die Klasse `TrainingSession` programmiert. Die folgende Tabelle zeigt die wichtigsten Methoden der Klasse `TrainingSession` und beschreibt diese.

Methode	Beschreibung
<code>train()</code>	Diese Methode startet das Training
<code>sortAndCalculateSum()</code>	Die Netze werden nach ihrer Bewertung sortiert und die beste und die durchschnittliche Bewertung wird ausgegeben.
<code>weightedRandomSelection()</code>	Ein zufälliges Netz wird zurückgegeben, allerdings ist es wahrscheinlicher, dass ein Netz mit besserer Bewertung zurückgegeben wird.
<code>randomizeArray()</code>	Die Netze werden zufällig angeordnet.
<code>isRunning()</code>	Testet ob trainiert wird.
<code>resetPlayer()</code>	Setzt die Fitnesswerte aller NNs zurück

4.3 Lernprozess/Optimierung

Natürlich kann das neuronale Netz aber Probleme nicht einfach ohne Hilfe oder zusätzliche Informationen lösen. Es muss, wie ein Mensch, erst einmal trainiert werden. Bei der Optimierung (Training) von neuronalen Netzen gibt es mehrere Möglichkeiten, von denen manche dem neuronalen Netz Beispiele geben und das neuronale Netz sich an diese annähert und sich damit auch einer allgemeinen Lösung annähert oder andere keine Beispiellösungen und im Bezug auf Brettspiele auch keine Strategie vorgeben. Wir haben uns bei unserem Projekt für einen evolutionären Ansatz entschieden, da auch hier keine Strategie vorgegeben wird. Bei unserer Optimierungsmethode werden mehrere Netze erzeugt, die alle gegeneinander Dame spielen. Nach jedem Spiel werden die beiden Netze bewertet, indem ihnen ein Fitnesswert zugeschrieben wird. Wenn alle Netze gegen alle Netze gespielt haben werden die mit den geringsten Fitnesswerten aussortiert und durch veränderte Versionen der anderen Netze ersetzt. Dieser Prozess kann nun sehr oft wiederholt werden. Auf diese Weise erhält man mit der Zeit immer bessere Neuronale Netze.

Bedingung	Punkte
Richtiger Zug	+100
Falscher Zug	-100
Schlag	+25

Abbildung 4.3: Fitnesspunkte für bestimmte Aktionen

Die zweite von uns implementierte Methode nutzt ein etwas unfreieres Lernen um einen schnelleren Fortschritt zu ermöglichen. Dabei spielen alle Netze gegen einen MiniMaxspieler. Sie werden auf die gleiche Weise bewertet wie auch bei der anderen Methode.

Die CPU Auslastung liegt bei einem Training ungefähr bei 8 % , da wir kein Multi-Threading in das Training implementiert haben. Das Programm nutzt also die zu Verfügung stehende Leistung schlecht und könnte theoretisch noch optimiert werden.

4.4 Zusammenfassung

Die Implementation des neuronalen Netzwerks selbst ist relativ einfach, da das neuronale Netz nur aus einer Abfolge von mathematischen Operationen und Veränderungen von Werten besteht und das Training mit unserem evolutionären Ansatz auch umsetzbar war. Beim Verbinden des neuronalen Netzes mit dem von uns implementierten Brettspiel Dame, standen wir aber am Anfang vor Schwierigkeiten, da es schwierig ist das richtige Layout und die richtige Trainingsumgebung zu programmieren und konfigurieren.

Die Leistung des neuronalen Netzes muss in mehrere Bereiche aufgeteilt werden. Da es Erstens eine gewisse Zeit Lernen muss, um die Regeln zu verstehen und sehr lange Lernen muss um Züge zu machen die als sinnvoll und gut bezeichnet werden können, es aber Zweitens für die Entscheidung nicht viel Zeit benötigt und es das Potenzial hat innerhalb von sehr kurzer Zeit einen sehr guten oder sogar perfekten Zug auszuwählen.

5. Andere Lösungsansätze

5.1 Datenbank als Lösungsansatz

Mit sehr hoher Rechenleistung ist es möglich das Spiel komplett zu berechnen, also jede einzelne Spielsituation mit dem besten Zug in eine Datenbank zu speichern. Mit dieser vollständigen Datenbank könnte man also theoretisch ein Spiel ohne Fehler spielen. Er wäre daher also unschlagbar.

Um das zu erreichen muss man einen kompletten Spielbaum erstellen. Jedoch steigt mit der Tiefe, wie wir es schon beim MiniMax erklärt haben der Aufwand nahezu exponentiell.

Um in dieser Datenbank auf die einzelnen Daten, also die Züge zugreifen zu können, bräuchte man einen Primärschlüssel, der in jedem Datensatz einzigartig ist. Diesen könnte man erhalten, indem man eine Zahl erstellt, die eine Spielsituation eindeutig identifiziert. Beispielsweise eine 32-Stelligen Zahl, bei welcher jede Stelle ein Feld auf dem Spielfeld repräsentiert. Alle Stellen dieser Zahl müssten dann mit bestimmten Ziffern für die einzelnen Zustände eines Feldes erstellt werden.

Diese Art der Speicherung wurde auch bei den .pfs Dateien in unserem Programm Checkers Simulation 2.0 verwendet, um Spielfeldsituationen zu speichern.

Neben dem Rechenaufwand ist auch der benötigte Speicher eine wichtige Komponente, die nicht außer Acht gelassen werden darf. Es wird angenommen, dass für die Kodierung UTF-8 benutzt wird.

Zunächst muss man die Größe eines Datensatzes bestimmen. Den Primärschlüssel kann man als `char`¹² betrachten und den Zug kann man, wie bei Checkers Simulation gezeigt wurde, mit vier Parametern speichern, also kann man auch hier jeden Parameter als einen `char` darstellen. (Mehrfachsprünge werden nicht betrachtet, da man nicht weiß wie viele es davon gibt und sie sowieso nur einen marginalen Bedeutung für das Ergebnis hätten.)

Dies wäre dann also die kleinstmögliche Größe eines Datensatzes.

$$32 \cdot 8 \text{ bits} + 4 \cdot 8 \text{ bits} = 256 \text{ bits}$$

Es ist möglich die Größe des Datensatzes mit der Anzahl der Situationen multiplizieren und in Terabyte umzurechnen.

$$\frac{256 \cdot 6,0111 \cdot 10^{20}}{(8 \cdot 10^{12})} \approx 1,924 \cdot 10^{12} \text{ TB}$$

Unsere Berechnungen für die Größe aller Spielfeldsituationen ist aber viel Größer als die Anzahl der möglichen Spielsituationen, also wird der richtige Wert um paar Stellen kleiner sein.

Trotz der starken Ungenauigkeiten in dieser Rechnung wird gezeigt, dass ein enorm hoher Speicherbedarf benötigt wird.

Insgesamt kann gesagt werden, dass diese Lösungsmöglichkeit mit Abstand die beste Spielstärke hätte. Dies geht aber auf Kosten des Speicher-, Rechen- und Implementationsaufwands.

5.2 Zufallsbasierter Algorithmus

Dieser Algorithmus, welchen wir als erstes zu Testzwecken implementierten, wählt durchgängig nur zufällige, aber mögliche Züge aus. Die Auswahl an richtigen Zügen erhält dieser Algorithmus aus unserem Programm Checker Simulation 2.0, welches, wie schon bereits erklärt, das Spielfeld nach allen möglichen Zügen durchsucht. Er ist in der Spielstärke oft sehr schwach, kann jedoch als ein Lösungsweg betrachtet werden, da er einen vollwärtigen Gegenspieler darstellt. Es besteht natürlich auch die Möglichkeit, dass er ein perfektes Spiel spielt jedoch ist dies Chance praktisch gleich null.

Insgesamt ist dieser Algorithmus von der Spielstärke her sehr schwach, hat jedoch den Vorteil, dass sowohl der Rechenaufwand als auch der Implementationsaufwand minimal ist.

¹² Ein `char` ist ein Java Datentyp. Er besteht aus einem Byte oder 8 Bits.

6. Vergleich/ Fazit

Nachdem wir die einzelnen Ansätze im Detail ausgearbeitet haben, folgt nun der Vergleich dieser bereits analysierten Aspekte.

Besonders die Spielstärke spielt eine wichtige Rolle, da diese deutlich macht, wie gut das Problem gelöst wurde. Wir haben sowohl beim MiniMax Algorithmus als auch beim Neuronalen Netz feststellen können, dass der Rechenaufwand mit der Spielstärke zusammenhängt. Jedoch lässt sich ein zentraler Unterschied bei dem Zeitpunkt und der Art der Rechenaufwands feststellen. Beim MiniMax entstehen diese vor jedem Spielzug, dadurch entsteht eine Verzögerung während des Spiels. Es wird somit nach jedem Zug nur ein Teil des ganzen Problems gelöst. Beim Neuronalen Netzwerk entsteht der Rechenaufwand durch das sogenannte Lernen des Netzes. Dabei entsteht aber keine Verzögerung während des eigentlichen Spiels, sondern es muss vorher eine sehr große Anzahl an Spielen gespielt werden. Der Vorgang der Lösungsfindung findet also grundsätzlich über einen größeren Zeitraum statt. Es lässt sich aber trotzdem sagen, dass der zeitliche Aspekt abhängig von der Spielstärke und der Rechenleistung der CPU ist. Das gilt grundsätzlich für jeden Ansatz. Ein extremes Beispiel dafür ist der Datenbank Ansatz.

Die "Gewichte" der Neuronen werden aber immer gespeichert. Somit bleiben die gelernten Informationen erhalten und müssen nicht erneut berechnet werden. Im Verhältnis zum Minimax-Algorithmus ist dies sehr effizient, denn bei diesem werden keine Informationen über einen längeren Zeitraum gespeichert, sondern alles muss immer wieder neu berechnet werden.

Der Minimax kann direkt auf einem vorher definierten bestimmten Niveau spielen, da die Spielstärke direkt durch die Tiefe angepasst wird. Beim NN kann die Spielstärke nicht direkt beeinflusst werden, da sie einfach nur langsam zunimmt.

Der Ressourcenverbrauch ist beim Minimax am geringsten, gefolgt von dem Neuronalen Netz, welches etwas mehr Speicher für die vielen Netze, die bei Training erzeugt werden, benötigt. Beides ist aber sehr gering im Vergleich zum Datenbank Ansatz, bei welchem die ganze Situation gespeichert werden müssen..

Alle Methoden sind relativ einfach zu implementieren, wenn man das grundlegende Prinzip verstanden hat und die Voraussetzung wie Rechenleistung, Zeit und Speicher besitzt. Beim Datenbank Ansatz hatten wir nicht alle Voraussetzungen und konnten ihn deshalb nicht implementieren. Wir haben aus eigener Erfahrung gesehen, dass das NN deutlich fehleranfälliger ist. Bei beiden Fehlerfindungen sind beide nicht einfach zugänglich.

Zusammengefasst lässt sich sagen, dass alle Ansätze Stärken und Schwächen aufweisen aber trotzdem in der Lage sind das Problem gut zu lösen. Der MiniMax ist deutlich flexibler, kann direkt eingesetzt werden und hat, im Gegensatz zum NN, einen genau definierbaren Rechenaufwand für eine bestimmte Spielstärke. Bei einer hohen Spielstärke und einer schwachen CPU können das Spielerlebnis durch die lange Berechnung aber negativ beeinflussen. Daher ist es sinnvoller in diesem Fall das NN zu verwenden. Dies benötigen nur Zeit im Vorhinein, dannach kann es ohne Einschränkungen auf fast jedem System verwendet werden.

Durch unsere Ergebnisse lassen sich allgemeine Unterschiede zwischen den Ansätzen feststellen, die auch auf andere Probleme abstrahiert werden können. Konventionelle Lösungsalgorithmen sind oft für simple klar definierbare Probleme die effizienteste Lösung wie auch beim Brettspiel Dame. Hingegen ist Maschinelles Lernen sinnvoll, wenn Dinge selbständig erlernen werden sollen, was bei dem Brettspiel auch vom Vorteil ist, da das Problem dann auf eine menschliche Weise gelöst werden kann und man vielleicht mehr das Gefühl hat gegen einen menschlichen Gegner zu spielen. Das NN hat ein besseres Verständnis von dem Problem und kann auch durch Mustererkennung ohne detaillierte Lösungsanweisungen oft komplizierte Probleme lösen, braucht aber viel Zeit für das Lernen. Dies haben wir bei unserem Projekt auch beobachten können. Datenbanken sind oft sinnvoll für große Datenmengen und es kann so die ganze Komplexität des Spiels gespeichert und wiedergegeben werden.

Trotz der Schwächen sind wir insgesamt mit den Ergebnissen der Facharbeit zufrieden. Wir haben fundamentale Unterschiede zwischen den einzelnen Ansätzen herausfinden können, die wir zuvor in unser Programm Checker Simulation 2.0 selbständig implementiert haben. Zudem haben wir vieles in den Bereichen maschinelles Lernen, objektorientierte Programmierung, GUI Programmierung, Netzwerk Programmierung und wissenschaftliches Arbeiten dazugelernt.

7. Quellenverzeichnis

https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden
https://www.focus.de/wissen/mensch/geschichte/erfindungen/tid-5356/rechenmaschine_aid_51261.html
<https://de.wikipedia.org/wiki/Pascaline>
https://de.wikipedia.org/wiki/Rechenmaschine#Historische_Entwicklung
https://de.wikipedia.org/wiki/Blaise_Pascal
https://de.wikipedia.org/wiki/Floating_Point_Operations_Per_Second
<https://de.wikipedia.org/wiki/TOP500>
www.de.wikipedia.org/wiki/Minimax-Regel
www.de.wikipedia.org/wiki/Minimax-Algorithmus
<https://de.wikipedia.org/wiki/Nervenzelle>

Bedienungsanleitung:

Installation:

Es muss einfach nur die .jar-Datei ausgeführt werden. Um Fehler zu vermeiden sollte die neueste Java version auf dem Computer installiert sein.

Optionen beim Start eines Spiels:

Wenn man auf Run>new Run klickt öffnet sich ein neues Fenster, in man einige verschiedenen Optionen auswählen kann:

- **runName:** Das spiel wird bennant. Ist nur notwendig, wenn gameRecording aktiviert ist.
- **gameRecording:** Wenn man diese Option anklickt, werden im Ordner resources/recordedGame einge Informationen über diese Spiele gespeichert, die für die Auswertung genutzt werden können.
- **Rounds:** Setzt die Anzahl der Runden, die gespielt werden sollen.
- **Spielerauswahl:** Ein Dropdown-Menü, indem alle Spieler angezeigt werden, die sich im resources/AI Ordner befinden + einen 'Player', der für einen menschlichen Spieler steht.
- **Display enabled:** Wenn diese Option ausgeschaltet wird, wird das Spiel nicht angezeigt.(Kann nur deaktiviert werden, wenn beide Spieler Computerspieler sind!)
- **using current playfield:** Spiel wird mit dem aktuellen Spielfeld gestartet(Kann nur aktiviert werden, wenn sich eine Situation auf dem Spielfeld befindet, die Beispielbar ist!)
- **Schieberegler:** Beeinflusst den Delay, den Computergegner haben(fast: 0sek, medium: 1sek, slow: 5sek).

Laden und Speichern von Spielsituationen:

Man kann Situationen auf dem Feld speichern, indem man Game>save situation benutzt. Eine Situation lässt sich aber nur mit Game>load situation laden, wenn gerade kein Spiel läuft. Dazu muss man im Auswahlfenster die gewünschte .pfs Datei auswählen. Man kann dann auch eine Spiel mit der geladen Ausgangssituation starten, indem man im Run>new Run Fenster useCurrentPf anklickt.

Während eines Spiels:

Das Spiel kann pausiert oder gestoppt werden unter run. Außerdem kann man unter preferences noch einige Optionen vom Start auch noch während einem Spiel verändern.

Wie man einen Spieler hinzufügen kann:

Wenn man einen Spieler selbstständig porgrammieren möchte, dann kann man diesen als .class oder .Jar Datei in den resources/AI Ordner einfügen. Eine der Klassen des Computerspielers benötigt dazu das Interface 'Player'. Die Information über das Spielfeld können über die öffentlichen getMethoden des Programmes erhalten werden.

NN Training:

Das NN Training Fenster kann geöffnet werden mit run>NN Training. Dort können die NN Einstellungen spezifisch angepasst werden und es kann eine Training gestartet und gestoppt werden. Die dabei entstandenen NN Dateien werden in dem Ordner resources/Trainingssessions gespeichert.

Um im Hauptporgramm gegen eines dieser trainierten NN's zu spielen, muss eine der .json Dateien, die ein NN repräsentieren, in den resources/NNPlayer1Info Ordner oder resources/NNPlayer2Info Ordner.

Weitere Funktionen:

Diese Funktionen haben keine große Relevanz für das Projekt, können aber trotzdem ganz nett sein. Beispielsweise kann bei preferences>Color die Farbe des Programmes und die Farbe der Schrift in der Konsole verändert werden. Über die Console kann man darüber hinaus auch weitere Dinge mithilfe von Commands einstellen.

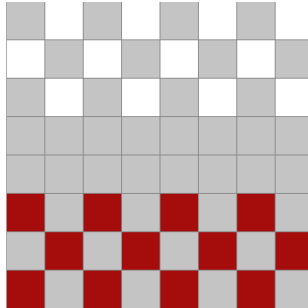
Außerdem hat das Programm Soundeffekte für das Spiel und einen Musicplayer. Diese können mit mit Preferences>Sound verändert werden. Lieder kann man im Ordner resources/Music einfügen.

Es gibt auch die Möglichkeit zwei Programme über ein lokales Netzwerk zu verbinden. So kann dann über die Konsole kommuniziert werden oder ein Spiel gestartet werden, bei dem die beiden gegeneinander spielen können.

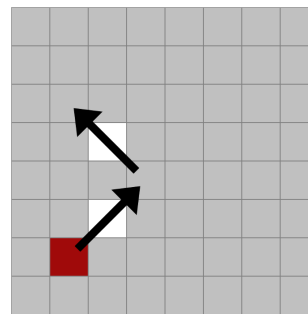
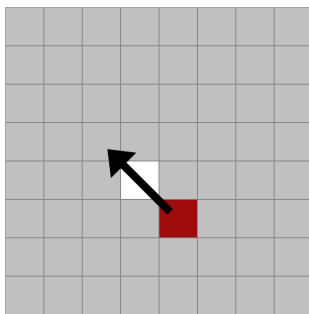
Regeln Checkers

Die von uns genutzte Variante nennt sich Checkers(bE) oder Draughts(aE) und wird vor in vielen englischsprachigen Ländern gespielt.

Ein 8×8 Spielfeld gespielt dient als Grundlage, wobei immer nur jedes zweite Feld bespielbar ist. Zu Beginn besitzt jeder Spieler 12 Figuren, die folgendermaßen auf dem Spielfeld angeordnet sind:



- Figuren können sich nur diagonal ein Feld vorwärts bewegen oder eine Figur schlagen.
- Es kann nur geschlagen werden, wenn diagonal links oder recht auf dem nächsten Feld ein gegnerische Figur steht und sich diagonal auf der Linie hinter dieser Figur eine freies Feld befindet.
- Das Schlagen ist zwingend. Wenn es eine Möglichkeit zu schlagen muss diese auch ergriffen werden.
- Wenn man einen Schlag vollführt, muss man direkt auf dem Feld hinter der geschlagen Figur landen.
- Es ist auch möglich mit einer Figur mehrere Schläge in einem Zug auszuführen. Auch hier ist das Schlagen der nächsten Figur zwingen notwendig.



- Wenn eine Figur geschlagen wird, dann wird sie permanent vom Spielfeld entfernt.
- Wenn man mit einer Figur die andere Seite des Spielfelds erreicht hat, wird die Figur zu einer Dame.(in Checker Simulation 2.0 wird das durch eine Krone auf der Figur repräsentiert). Diese hat nun auch die Möglichkeit rückwärts zu gehen und rückwärts zu schlagen.
- Es fängt immer der Spieler mit den roten Figuren an.
- Das Ziel des Spiels ist entweder alle Figuren des Gegner vom Spielfeld zu entfernen oder die noch verbleibenden gegnerischen Figuren zugunfähig zu machen.
- Wenn 50 Züge hintereinander keine Figur mehr geschlagen wird, dann wird automatisch ein Unentschieden erzwungen. Beide Spieler können sich auch schon vorher auf ein Unentschieden einigen.