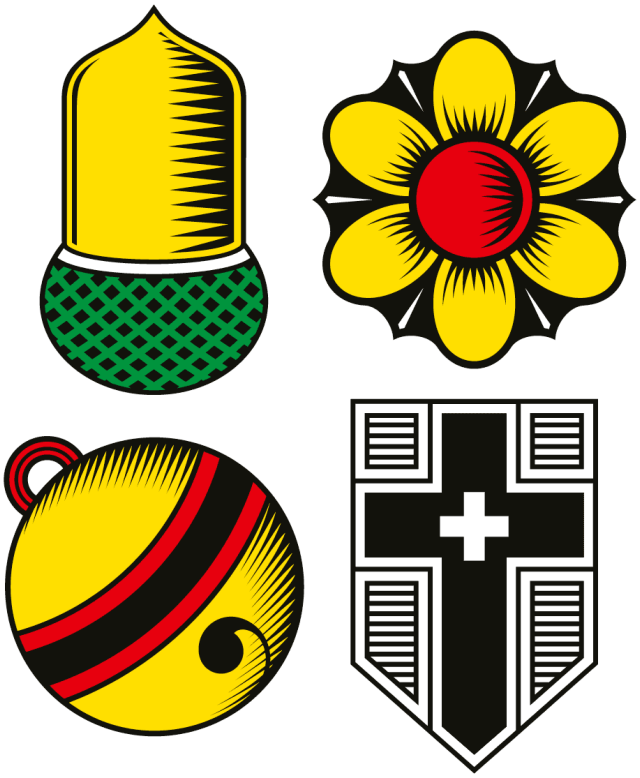
Liceo Artistico, Kantonsschule Freudenberg Zürich
Schuljahr 2023/24



Jassen ohne Intuition

PROGRAMMIEREN UND ANALYSIEREN VON UNTERSCHIEDLICHEN SUCHVERFAHREN ANHAND DES DIFFERENZLERS

Tamara L. Burri | Klasse L5b2

Betreuende Lehrpersonen: Dr. Patric Müller & Pavel Lunin

Zusammenfassung

In dieser Arbeit erforsche ich vier Suchalgorithmen für Multiplayer-Spiele, basierend auf dem Differenzler. Durch den Verzicht auf Machine-Learning möchte ich herausfinden, ob Jassen auch ohne Erfahrung möglich ist. Dabei stellt die imperfekte Information des Spiels eine grosse Herausforderung dar. Ich entwickle spezifische Methoden zur Modellierung von Spielsituationen, um diese Schwierigkeit zu überwinden. Die statistische Auswertung der durchschnittlichen Differenzen zeigt, dass die Suchalgorithmen besser abschneiden als zufällige Spiele. Allerdings erreichen sie nicht das Niveau von ProfispielerInnen.

Vorwort

Egal wohin ich auch gehe, es wird immer ein Kartenset mitgenommen. Manchmal spiele ich mit meinen FreundInnen Shithead, Arschlöchle-Jass, oder Klassiker wie UNO. Dabei sind Kartenspiele nicht nur ein guter Zeitvertrieb während der Pause, sondern in ihrer Essenz unglaublich faszinierend.

Man stelle sich vor: Ein normales Deck von 52 Karten kann auf 8×10^{67} Arten gemischt werden. Das bedeutet, dass es mehr Blattreihenfolgen gibt als Sand am Strand oder als Atome auf unserer Erde. Mischt man ein Kartenset, so erhält man eine einzigartige Kartenreihenfolge, die es noch nie zuvor in der Menschheitsgeschichte gegeben hat und wahrscheinlich nie geben wird – und das (fast) jedes Mal.

Mit jedem Mal Karten Mischen und mit jeder getroffenen Entscheidung bewegen wir uns innerhalb undenkbarer Grössenordnungen, dabei können wir uns nur einen Bruchteil aller Möglichkeiten vorstellen. So ist es für uns unmöglich, die langfristigen Konsequenzen unseres Handels vorauszusehen. Wir sind gezwungen, unserem Bauchgefühl zu vertrauen, wohingegen Computer in der Lage sind, innert wenigen Sekunden Millionen von Positionen zu analysieren und den besten Zug zu finden.

Es ist sowohl das unsichtbare Potential von Spielen als auch die Art, wie MathematikerInnen und InformatikerInnen dieses Potential ausschöpfen, das mich fasziniert. Aus diesem Grund war schon von Anfang an für mich klar, dass meine Maturarbeit Karten- oder Brettspiele behandeln soll. Dabei wollte ich mich hauptsächlich auf den theoretisch-mathematischen Analyse von Spielen fokussieren, und anhand dieser Grundlagen ein Computerprogramm schaffen, das selbst in der Lage ist, Spiele zu spielen.

Inhaltsverzeichnis

1. Einleitung

Künstliche Intelligenzen treffen tagtäglich Entscheidungen und beeinflussen unser Leben auf eine Art und Weise, wie wir es uns kaum vorstellen können. Sie schlagen uns Content auf Netflix vor, betreiben Risikomanagement, bestimmen unsere Kreditwürdigkeit oder beschliessen, wer ans Bewerbungsgespräch kommt und wer nicht. Doch wie schaffen es Computer, intelligent zu handeln?

Die Ursprünge der Künstlichen Intelligenz reichen bis ins Jahr 1951 zurück, als die zwei Informatiker Christopher Strachey und Dietrich Prinz die ersten funktionsfähigen KI-Programme entwickelten. Sie konnten Schach und Dame spielen, was zu dieser Zeit eine revolutionäre Leistung darstellte (Copeland, 2008). Brettspiele erwiesen sich als ideales Testgelände sowohl für das Entwickeln von KIs als auch für die Analyse von Entscheidungen. Sie sind gewissermassen ein vereinfachtes Modell der echten Welt: Mehrere SpielerInnen setzen sich an einem Tisch, alle mit dem gegensätzlichen Ziel, zu gewinnen. Das zwingt sie, die möglichen Gewinne und Verluste abzuwägen und ökonomisch zu handeln.

Die Wahl von Schach als Forschungsgebiet war aber keineswegs ein Zufall, denn Schach ist mathematisch lösbar. Das bedeutet, dass es in jeder erdenklichen Spielsituation eine beste Handlungsoption gibt. (Zermelo, 1913) Das perfekte Spiel eines Computers führt zwangsläufig zu einem Sieg gegen einen Menschen. So gelang es dem Schach-Computer Deep-Blue im Jahre 1997 zum ersten Mal, den Schachmeister Garry Kasparov zu schlagen. (IBM)

In der realen Welt gibt es aber nur selten eine mathematisch definierte «beste» Option. Deshalb war es schwierig, die anfänglichen Erfolge von Schachcomputern auf andere, komplexere Bereiche zu übertragen. Der Mathematiker John von Neumann interessierte sich aus diesem Grund besonders für das Glückspiel Poker; man kennt zwar die eigenen Karten, aber nicht jene der GegnerInnen. Eigene Entscheidungen müssen deshalb auch auf dem Verhaltensmuster der MitspielerInnen basieren. Solche KIs basieren weniger auf robuste theoretische Prinzipien, sondern auf heuristische Methoden. (Romer, 2022)

In dieser Hinsicht ähnelt das Jassen und insbesondere der Differenzler dem Poker. Auch wenn das Spiel nach 36 Zügen endet, entsteht durch die zufällige Verteilung der Karten, durch die Anzahl SpielerInnen und durch die Punktansage am Anfang eine unberechenbare Unsicherheit. Trotzdem hat das Jassen bisher weniger Aufmerksamkeit erfahren und ist praktisch unerforscht geblieben. Daher erschien mir der Differenzler als ein spannendes Thema für meine Maturarbeit.

1.1. Leitfragen

Das Hauptziel meiner Arbeit war es, die Fähigkeiten und Grenzen von künstlichen Intelligenzen anhand des Jassspiels, speziell des Differenzlers, zu untersuchen. Ich wollte folgende Fragen beantworten:

1. Kann ich trotz der Komplexität des Spiels eine funktionierende KI entwickeln?
2. Wie sollen KIs ihre Entscheidungen treffen und ihre Ansagen bestimmen?
3. Wie analysiere ich die Ergebnisse?
4. Wie gut schneiden die KIs im Vergleich zu einer rein Zufälligen Strategie ab?
5. Wie gut schneiden die KIs untereinander ab?
6. Wie gut schneiden die KIs im Vergleich zu Menschen ab?

1.2. Aufbau der Arbeit

Es war mir wichtig, meine Arbeit für möglichst alle zuganglich zu gestalten, auch für Personen, welche keine Erfahrung mit dem Jassen, der Spieltheorie oder dem Programmieren haben. Die grundlegenden Begriffe und Konzepte werde ich im Verlauf meiner Arbeit erklären, um ein umfassendes Verständnis meines Projekts zu ermöglichen. Code-Ausschnitte werden in Form von Pseudocode präsentiert und sind praktisch wie normaler Text zu lesen.

Im ersten Kapitel werde ich Spielregeln des Differenzlers erläutern und hervorheben, was dieses Kartenspiel auszeichnet. Außerdem werde ich einige grundlegende Jass-Begriffe vorstellen, die im weiteren Verlauf meiner Arbeit Verwendung finden.

Im nächsten Kapitel gebe ich eine Einführung in die Spieltheorie, welche es erlaubt, Spiele auf abstraktere Weise zu analysieren. Dabei werde ich erläutern, unter welchen Bedingungen ein Spielverlauf vorhersehbar ist und wie das Minimax-Verfahren genutzt werden kann, um einen besten Spielzug zu finden. Ich werde mich dabei auf intuitive Erklärungsansätze konzentrieren und formale Schreibweisen weitgehend vermeiden.

Anschließend werde ich erklären, wie Suchalgorithmen in der Lage sind, gute von schlechten Entscheidungen zu unterscheiden. Dabei werde ich auf die zuvor erarbeiteten Methoden und Konzepte der Spieltheorie, insbesondere dem Minimax und dem Nash-Gleichgewicht, zurückgreifen. Ich werde die Datenstruktur des Spielbaums erläutern und wie der Minimax-Algorithmus diese rekursiv durchläuft. Des Weiteren werde ich erklären, wie Suchalgorithmen bei Mehrspielerspielen funktionieren.

Im darauffolgenden Kapitel zeige ich, wie ich das Wissen aus den vorherigen Kapiteln in die Praxis umgesetzt habe. Dabei werde ich die Struktur meines Programms und die Herausforderungen beschreiben, die ich während des Entwicklungsprozesses bewältigen

musste. Zusätzlich werde ich das Benutzerprogramm vorstellen, mit dem SpielerInnen gegen die KIs antreten können.

Abschließend führe ich automatisierte Tests mit den KIs durch, um zu analysieren, wie gut sie abschneiden. Ich werde die Ergebnisse untereinander vergleichen, aber auch mit Ergebnissen aus Jass-Turnieren. Auch werde ich die direkte Konfrontation von Menschen und Computer Analysieren.

Am Ende werde ich reflektieren, inwieweit ich meine Ziele erreicht habe und was ich möglicherweise hätte besser machen können.

2. Jass und der Differenzler

Jass ist der Übergriff für eine Reihe von unterschiedlichen Schweizer Kartenspielen, dabei hat jede Variante ihre eigenen Abweichungen, doch die Grundregeln sind wie folgt:

- Es wird mit 36 Karten gespielt (A, K, Q, J, 10, 9, 8, 7, 6 in je 4 Farben).
- Am Anfang des Spiels wird Trumpf (siehe 2.1) ausgewählt.
- Jede Karte hat einen bestimmten Punktewert (siehe Tabelle 1).
- Ziel des Spiels ist es, eine gewisse Punktzahl zu erreichen.

Karte (Trumpf)	Punktewert	Karte (normal)	Punktewert
Under / Bauer (J)	20	Ass (A)	11
Nell (9)	14	König (K)	4
Ass (A)	11	Ober/Dame (Q)	3
König (K)	4	Under / Bauer (J)	2
Ober/Dame (Q)	3	Zehn (10)	10
Zehn (10)	10	Neun (9)	0
Acht (8)	0	Acht (8)	0
Sieben (7)	0	Sieben (7)	0
Sechs (6)	0	Sechs (6)	0

Tabelle 1: Karten und ihre Punktewerte, absteigend sortiert nach Stärke. Trumpf sticht immer.

2.1. Jass-Begriffe

Im Verlaufe des Kapitels werden einige typische Jass-Begriffe erwähnt, die im Folgenden kurz erklärt werden.

Stich Pro *Stich* legt jede Person eine Karte. Die stärkste Karte, die gespielt wurde, *sticht* und die Person, die sie gelegt hat, *gewinnt den Stich*. Sie kann den Punktewert alle Karten zu ihren Gunsten aufschreiben und die Karten beiseitelegen.

Abstechen bedeutet, den Stich einer anderen Person zu verhindern.

Bock Als *Bock* bezeichnet man die stärkste Karte ihrer Farbe. Ein Beispiel: Rosen Ass ist nicht mehr im Spiel, das bedeutet, dass Rosen König der neue *Bock* ist[1].

Farbe Die Farbe der ersten Karte bezeichnet man als *ausgespielte Farbe*, die übrigen nennt man *Nebenfarben*.

[1] Man geht hierbei davon aus, dass Rose nicht Trumpf ist.

Bedienen bedeutet, eine beliebige Karte der *ausgespielten Farbe* zu spielen.

Trumpf Über- und untertrumpfen bedeutet, einen höheren, bzw. tieferen Trumpf auszuspielen, als bereits gespielt wurde.

2.2. Was den Differenzler auszeichnet

Unter allen Jass-Arten ist der Differenzler dafür bekannt, die fairste Variante zu sein, da nicht das eigene Blatt, sondern hauptsächlich das eigene Können eine Rolle spielt (Muff, 2022a). Das liegt daran, dass am Anfang der Runde alle SpielerInnen *ansagen,* beziehungsweise schätzen, wie viele Punkte sie mit ihrer gegebenen Hand machen werden. Die Ansage kann je nach Vereinbarung entweder offen oder verdeckt sein.

Nach der Ansage beginnt dann das eigentliche Spiel:

1. SpielerInnen legen im Gegenuhrzeigersinn eine Karte auf den Jassteppich, wobei ihre Zugmöglichkeiten zum Teil eingeschränkt sind (siehe 2.2.1.)
2. Nachdem alle SpielerInnen ihre Karte gespielt haben, wird gestochen.
3. Der Prozess wird so oft wiederholt, bis alle Karten gespielt wurden. (In einer Runde mit vier SpielerInnen folglich 9 Stiche).
4. Der letzte Stich zählt weitere 5 Punkte.

Die Person mit der tiefsten Differenz zwischen angesagter und tatsächlicher Punktzahl gewinnt. Demnach können auch schlechte Karten zum Sieg verhelfen.

2.2.1. Spielbarkeit von Karten

Wie vorhin angedeutet, können nicht alle Karten in jedem Zeitpunkt gespielt werden. Im Differenzler gilt der *eingeschränkte Farbzwang*:

- Die erste Person eines Stiches hat all ihre Karten zur Auswahl.
- Die anderen SpielerInnen müssen (wenn möglich) die ausgespielte Farbe spielen.
- Wer nicht bedienen kann, hat alle Karten zur Auswahl.
- Untertrumpfen darf man nur, wenn die ausgespielte Farbe Trumpf ist oder nicht bedient werden kann.
- Ansonsten dürfen Trümpfe immer gespielt werden, unabhängig von der ausgespielten Farbe.

- Bauer Trumpf ist vom Spielzwang ausgeschlossen.

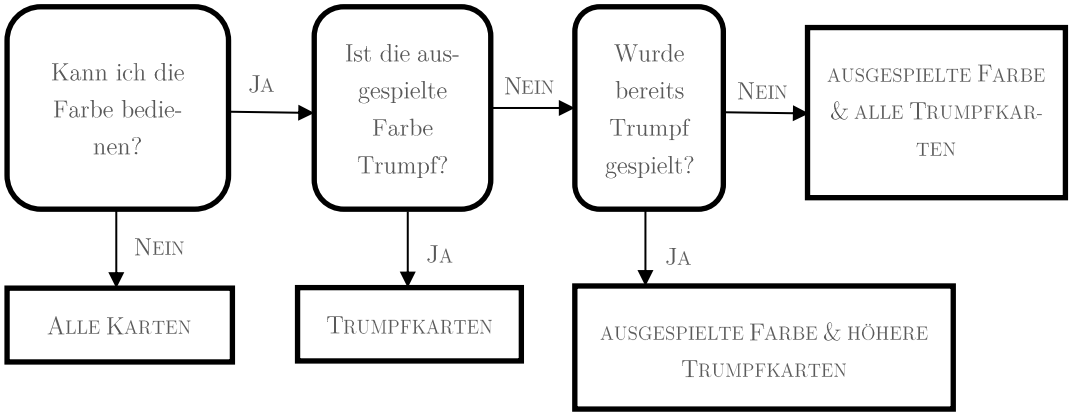


Abbildung 1: Flussdiagramm zur Bestimmung spielbarer Karten. Vom Trumpf Bauer ist abzusehen.

3. Entscheidungs- und Spieltheorie

Die Entscheidungs- bzw. Spieltheorie ist ein Teilgebiet der Mathematik, das insbesondere in der Ökonomie und in der Soziologie Anwendung findet. Sie versucht anhand abstrakter Modelle die Entscheidungen verschiedener AkteurInnen zu analysieren und zu werten, wobei in der Entscheidungstheorie vor allem unkontrollierbare äussere Einflüsse von Bedeutung sind, während in der Spieltheorie die Handlungen anderer SpielerInnen im Fokus liegen.

Im Rahmen dieser Arbeit ist die nicht-kooperative Spieltheorie relevant. Es gilt die Annahme, dass die TeilnehmerInnen alle rational, strategisch und egoistisch handeln, um das Spiel zu ihren Gunsten zu lenken. Ausserdem geht man meistens von *vollständiger Information* aus[2]. Das bedeutet, dass alle SpielerInnen das Verhaltensmuster ihrer GegnerInnen kennen.

3.1. Was ist ein Spiel?

Diese Frage mag erstmals trivial erscheinen, doch ohne sie wäre die Spieltheorie undenkbar, denn Mikado ist nicht gleich Monopoly und Solitaire ist nicht gleich Schach. Damit eine spieltheoretische Analyse möglich ist, muss ein Spiel folgende Bedingungen erfüllen (Staffin, 1996):

- Es gibt mindestens zwei TeilnehmerInnen.
- Alle TeilnehmerInnen haben eine Anzahl unterschiedlicher Strategien bzw. Handlungsoptionen.
- Die unterschiedlichen Handlungen beeinflussen den Spielverlauf.
- Jeder Spielausgang hat ein numerisches Ergebnis für alle TeilnehmerInnen, auch genannt *Pay-off*: Je vorteilhafter die Handlung, desto grösser der Pay-off.

Ein **Spiel** im Sinne der Spieltheorie ist ein mathematisches Modell zur Beschreibung von Vorgängen, in denen mehrere TeilnehmerInnen gegenseitig die Ergebnisse ihrer Entscheidung beeinflussen. (Ortmanns & Albert, 2008)

[2] nicht zu verwechseln mit *perfekter Information*

3.1.1. Quellen der Ungewissheit

Im Idealfall lässt sich der Handlungsverlauf eines Spiels vollständig voraussagen, es gibt jedoch drei Quellen der Ungewissheit, welche ein Spiel zwar spannend und unterhaltsam, aber auch unberechenbar machen können.

Abbildung 2: Die drei Ursachen der Ungewissheit in Gesellschaftsspielen: Gewonnen wird mit Glück, Logik und Bluff. (Bewersdorff, 1998)

Zufall tritt meistens in Form von Würfeln oder gemischten Spielkarten auf. In Glücksspielen ist er der dominierende Aspekt.

Kombinatorik ist vertreten in Spielen wie Schach oder Dame, hier sind Zugmöglichkeiten durch die Spielregeln fixiert. Die Anzahl der möglichen Zugkombinationen kann für Menschen unüberschaubar sein, was die Vorhersage der Konsequenzen erschwert. Dieses Phänomen wird oft als "kombinatorische Explosion" bezeichnet (siehe Abbildung 3).

Strategie, bzw. fehlende Information sind in Spielen wie Schere-Stein-Papier oder Poker präsent. Im Falle von Poker haben die SpielerInnen unterschiedliche Informationsstände; sie wissen nur ihre eigenen Karten und können höchstens die bereits gespielten Karten zählen. In Schere-Stein-Papier entsteht die Ungewissheit dadurch, dass beide SpielerInnen gleichzeitig eine Entscheidung treffen, und so nicht wissen können, was die andere Person wählen wird. In solchen Fällen spielt der Bluff eine wichtige Rolle.

Der Jass enthält alle drei Quellen der Ungewissheit (siehe Abbildung 2) Im Differenzler spielt der Zufall jedoch nur eine untergeordnete Rolle, da gute oder schlechte Karten im Grunde keinen Einfluss auf die Gewinnchancen haben. Viel wichtiger ist hingegen die Strategie: Das Spiel verfügt über *imperfekte Information*, heisst also, dass die SpielerInnen am Anfang nur ihre eigenen Karten kennen, und anhand ihrer Karten eine Schätzung machen müssen. Dabei haben sie erst im Verlaufe des Spiels die Möglichkeit einzugrenzen, was die Karten ihrer GegnerInnen sind. Die Kombinatorik ist ebenfalls ein wichtiger Aspekt des Spiels: SpielerInnen haben im ersten Stich bis zu neun Karten zur Auswahl, dann acht, dann sieben usw. Das bedeutet, dass im Extremfall bis zu $9!^4 = 1.73 \times 10^{22}$ potenzielle Spielverläufe gibt[3].

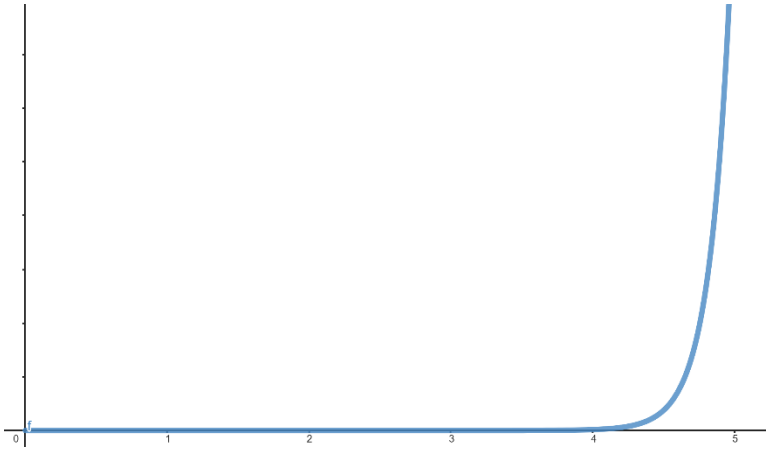


Abbildung 3: Der Graph von $f(x) = x!^4$ zeigt, wie schnell die Anzahl der möglichen Spielverläufe steigt. (Kombinatorische Explosion)

3.1.2. Die Berechenbarkeit eines Spiels

Die Frage, welche Bedingungen ein Spiel in der Theorie berechenbar machen, beschäftigte bereits den deutschen Mathematiker und Spieltheorie-Pionier Ernst Zermelo im Jahre 1913. Er ging von Schachrätseln aus, in denen eine Spielsituation gegeben ist und in einer bestimmten Anzahl von Zügen ein Schachmatt erzwungen werden soll. Wenn im Endspiel ein Ausgang erzwungen werden kann, wieso sollte das nicht für das gesamte Spiel gelten? Zermelo formulierte seine Frage wie folgt:

> *«Kann der Wert einer beliebigen, während des Spiels möglichen Position für eine der spielenden Parteien sowie der bestmögliche Zug mathematisch-objektiv bestimmt, oder wenigstens definiert werden, ohne dass auf solche mehr subjektiv-psychologischen wie die des «vollkommenen Spielers» und dergleichen Bezug genommen zu werden bräuchte?»* (Zermelo, 1913)

[3] (Wobei die tatsächliche Zahl eher bei 2.42×10^{16} liegt, siehe Kapitel 5.2.5).

Er konnte beweisen, dass jede Position ein eindeutig bestimmtes Spielergebnis und somit auch einen besten Spielzug besitzt. Dasselbe gilt für jedes Spiel, dass folgende fünf Bedingungen erfüllt (Zermelo, 1913):

- **Das Spiel wird von zwei Personen gespielt.**
- **Es ist ein Nullsummenspie**l. (Gewinn des einen Spielers ist gleich Verlust des anderen, die Summe aller Pay-offs ist somit immer Null)
- **Das Spiel ist endlich.** (es gibt endlich viele Zugmöglichkeiten und das Spiel ist nach einer begrenzten Anzahl von Spielzügen zu Ende)
- **Das Spiel hat perfekte Information.** (Alle Spieler sind über den gesamten Spielstand informiert)
- **Es gibt keine zufälligen Einflüsse.**

Da der Jass nicht alle dieser Bedingungen erfüllt, werden zur Einführung der wichtigsten Konzepte in dieser Arbeit Zweispieler-Nullsummenspiele untersucht.

3.2. Spieltheoretische Methode

Der Kern der Spieltheorie ist die Abstrahierung von komplexen Spielsituationen mithilfe einfacher Darstellungsarten, wie zum Beispiel in der Tabelle 2. Die zwei SpielerInnen Amelie und Beatrice spielen gegeneinander Schere-Stein-Papier:

A \ B	Schere	Stein	Papier
Schere	(0, 0)	(1, 1)	(1, –1)
Stein	(1, –1)	(0, 0)	(–1, 1)
Papier	(–1, 1)	(1, –1)	(0, 0)

Tabelle 2: Spielmatrix einer Runde Schere-Stein-Papier. Jeder Ausgang hat einen numerischen Pay-off (1 für einen Sieg, 0 für Unentschieden, -1 für Niederlage) für sowohl Amelia als auch Beatrice (A, B).

Man kann anhand der Tabelle die Pay-offs der unterschiedlichen Entscheidungen einsehen: Wählen beide Schere, ergibt sich ein Unentschieden $(0, 0)$. Wählt Amelie Stein und Beatrice Papier, so gewinnt Beatrice und der Pay-off lautet $(-1, 1)$.

In diesem Fall handelt es sich um ein Nullsummenspiel (siehe 3.1.2), und somit gilt $(A = -B)$. Bei Nullsummenspielen kann die Darstellung vereinfacht werden, indem man nur den Pay-off von A berücksichtigt.

A \ B	Schere	Stein	Papier
Schere	0	–1	1
Stein	1	0	–1
Papier	–1	1	0

Tabelle 3: Vereinfachte Darstellung der Tabelle 2. Der Pay-off von Amelie ist der negative Payoff von Beatrice $(A = -B)$.

Positive Werte sind vorteilhaft für Amelie und negative Werte sind vorteilhaft für Beatrice. Das Ziel von Amelie ist es demnach, ihren eigenen Payoff zu maximieren, wohingegen Beatrice versucht, den Payoff Amelies zu minimieren. Wenn beide die Entscheidungen der anderen Person vorhersagen könnten, gäbe es eine unendliche Abfolge von Strategieänderungen: A wählt Schere, B wählt Stein, A wählt Papier und so weiter.

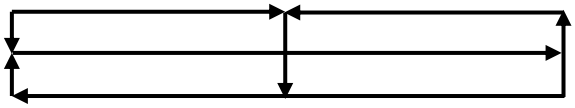
A \ B	Schere	Stein	Papier
Schere Stein Papier			

Tabelle 4: Pfeile stellen den Wechsel zur besten Strategie der Tabelle 3 dar.

Die beste Strategie in diesem Fall wäre, die Entscheidungen gleichmäßig zu verteilen. Wenn Amelie mit einer Wahrscheinlichkeit von 33 Prozent eine der drei Optionen wählt, kann sie einen Vorteil gegenüber Beatrice erlangen, die möglicherweise ungleichmäßig entscheidet. Dies wird als gemischte Strategie bezeichnet.

Wenn beide dieselbe gemischte Strategie wählen, werden sie im Laufe der Zeit den gleichen Punktestand erreichen. Unentschieden ist daher der Gleichgewichtszustand des Spiels.

3.2.1. Das Nash-Gleichgewicht

Die oben beschriebene Situation ist nicht immer gegeben. In einigen Spielen hat ein Amelie bessere Strategien zur Verfügung als Beatrice und kann ein Endresultat erzwingen. In solchen Fällen werden beide SpielerInnen ständig zwischen den besten Strategien wechseln, bis ein Gleichgewichtszustand erreicht wird.

A \ B	Option 1	Option 2	Option 3	Option 4
Option 1	12	–1	1	0
Option 2	5	1	7	–20
Option 3	3	2	4	3
Option 4	–16	0	0	16

Tabelle 5: Spielmatrix einer beliebigen Situation, in der sowohl Amelie als auch Beatrice 4 unterschiedliche Optionen zur Auswahl haben. (Staffin, 1996)

In dieser Situation hat Amelie offensichtlich einen Vorteil, wenn sie Option 3 wählt, da sie damit 2 Punkte sicherstellt. Beatrice hat weniger vorteilhafte Optionen und muss hoffen, dass Amelie riskante Entscheidungen trifft. Wenn Amelie jedoch rational handelt, wird sie Option 3 wählen und Beatrice wird gezwungen sein, Option 2 zu wählen, um ihren Verlust zu minimieren.

Ähnlich wie in Tabelle 3 lässt sich auch hier der Wechsel zur besten Strategie grafisch darstellen:

A \ B	Option 1	Option 2	Option 3	Option 4
Option 1				
Option 2				
Option 3				
Option 4				

Tabelle 6: Der Wechsel zur besten Strategie grafisch dargestellt. (Staffin, 1996)

Wie zuvor möchte Amelie ihren eigenen Punktestand maximieren und Beatrice möchte ihn minimieren. Sie werden ständig zur besseren Strategie wechseln. Schließlich werden sie garantiert Option 3 bzw. Option 2 wählen, unabhängig von der Ausgangssituation (in Tabelle 6 blau markiert). Dies wird als gegenseitige beste Antwort bezeichnet. Sobald dieser Zustand erreicht ist, lohnt es sich für keinen der beiden SpielerInnen, von ihrer Strategie abzuweichen, da dies ihren eigenen Pay-off verschlechtern würde. Diesen Zustand nennt man auch das Nash-Gleichgewicht, benannt nach dem Mathematiker John F. Nash, der bewiesen hat, dass es in jedem strategischen Spiel mindestens ein Nash-Gleichgewicht geben muss (Bärtschi, 2011).

Das **Nash-Gleichgewicht** ist das Resultat der gegenseitigen besten Antwort. Für keine der TeilnehmerInnen lohnt es sich, von der gewählten Strategie abzuweichen. Für jedes strategische Spiel gibt es ein Nash-Gleichgewicht.

3.2.2. Minimax

Das Nash-Gleichgewicht muss nicht zwingend mithilfe eines aufwendigen Bewegungsdiagramms wie in Tabelle 6 bestimmt werden. Es kann auch das Prinzip des *Minimaxes* angewendet werden.

B / A	Option 1	Option 2	Option 3	Option 4	Min	Maximin
Option 1	12	–1	1	0	–1	
Option 2	5	1	7	–20	–20	
Option 3	3	2	4	3	2	2
Option 4	–16	0	0	16	–16	
Max	12	2	7	16		
Minimax		2				

Tabelle 7: Ermittlung des Nash-Gleichgewichts mithilfe des Minimax-Prinzips.

Der Minimax ist der kleinste der maximalen Werte, während der Maximin der größte der minimalen Werte im Spiel ist. Im vorliegenden Beispiel beträgt der grösste Maximalwert für Amelie 16 Punkte, und der kleinste 2 Punkte. Beatrice versucht natürlich, diesen Wert zu minimieren. Sie wird daher den Minimax-Zustand mit 2 Punkten erzwingen.

Das Nash-Gleichgewicht entsteht durch die Bedingung, dass der Minimax-Wert gleich dem Maximin-Wert ist. Anders ausgedrückt: Das Nash-Gleichgewicht ist der kleinste Wert in einer Zeile (MIN) und gleichzeitig der größte Wert in einer Spalte (MAX). Dieses Konzept lässt sich auch als Sattelpunkt darstellen, da er sowohl für Amelie als auch für Beatrice die optimale Antwort darstellt.

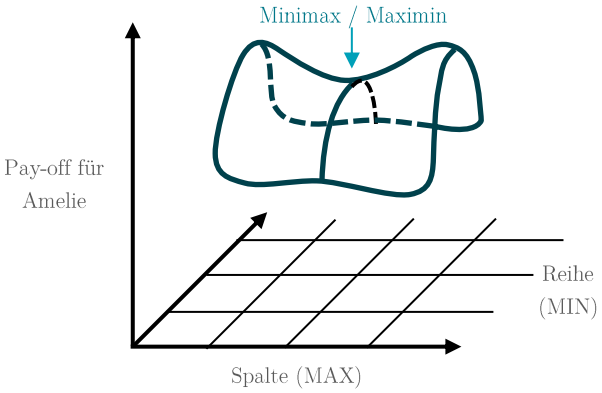


Abbildung 4: Grafische Darstellung des Sattelpunktes. (Staffin, 1996)

Der **Minimax** ist der **Sattelpunkt** einer Matrixdarstellung. Er ist der grösste Punkt einer Spalte und gleichzeitig der kleinste Punkt einer Reihe. In Nullsummenspielen ist der Minimax dasselbe wie das **Nash-Gleichgewicht**.

4. Künstliche Intelligenz

Das Ziel einer künstlichen Intelligenz besteht darin, Entscheidungen zu treffen, die idealerweise zu einem bestmöglichen Spielausgang führen. Es gibt verschiedene Herangehensweisen, wie eine KI dies erreichen kann. Eine Möglichkeit ist das maschinelle Lernen (Englisch: Machine Learning) und die Verwendung neuronaler Netzwerke. Hier lernt ein Computer, Muster zu erkennen und basierend auf vergangenen Spielsituationen sowie eigener "Erfahrung" gute Entscheidungen zu treffen. Die KI ist flexibel und kann sogar die Spielregeln selbst erkennen, um unterschiedliche Strategien zu entwickeln. Allerdings erfordert diese Methode enorme Mengen an Trainingsdaten und Rechenleistung.

Im Rahmen dieser Arbeit werden ausschließlich hartkodierte KIs behandelt. Hartkodierte KIs können nicht lernen, da sie einer festen Abfolge von Anweisungen (einem *Algorithmus*) folgen, die nicht verändert werden können. Der Computer hat kein echtes Verständnis der Spielregeln und kann nicht lernen. Daher eignen sich hartkodierte KIs besonders für Situationen, in denen die Rahmenbedingungen konstant sind, wie im Schachspiel. Die Leistungsfähigkeit dieser KI hängt hauptsächlich von der Qualität des geschriebenen Codes und der Rechenleistung des Computers ab.

Der Vorteil einer hartkodierten KI ist die Nachvollziehbarkeit ihrer Entscheidungen. Neurale Netze gleichen hingegen mehr einer Blackbox. Die inneren Prozesse sind oft zu komplex, die Anzahl Parameter zu unübersichtlich und die Resultate nur schwer zu reproduzieren, sodass die Entscheidungen eines neuralen Netzwerks unmöglich zu verstehen sind.

4.1. Einführung in Algorithmen

Ein *Algorithmus* ist ähnlich wie ein Rezept in einem Kochbuch: eine endliche Reihe von Anweisungen, die von einem Menschen oder einem Computer ausgeführt können, um ein spezifisches Problem zu lösen. Ein einfaches Beispiel ist die Überprüfung, ob eine Zahl durch 3 teilbar ist, indem man die Quersumme der Zahl berechnet und überprüft, ob sie durch 3 teilbar ist. Algorithmen können jedoch auch sehr komplex sein, wie bei einem Navigationssystem, das die kürzeste Route von A nach B finden muss.

Der Euklidische Algorithmus ist einer der ersten bekannten Algorithmen. Er dient dazu, den größten gemeinsamen Teiler (ggT) zweier Zahlen zu ermitteln und ist rekursiv. Das bedeutet, dass der Algorithmus Teil seiner eigenen Definition ist, indem derselbe Vorgang wiederholt angewendet wird, bis ein endgültiges Ergebnis erzielt wird.

Seien a und b natürliche, nicht negative Zahlen und $a > b$:

Schritt 1: Dividiere a durch b.

Schritt 2: Wenn kein Rest entsteht, so ist der grösste gemeinsamer Teiler b.

Wenn der Rest $r = 1$ ist, so ist der ggT $= 1$.

Wenn $r \neq 0$ & $r \neq 1$, ersetze b durch r und kehre zu **Schritt 1** zurück (Rekursion)

Diese Anweisungen können auch in eine für Computer verständliche Programmiersprache übersetzt werden.

```
Funktion ggT(a, b)

       Rest = a : b

       Falls Rest = 0

              Ergebnis = b

       Falls stattdessen Rest = 1

              Ergebnis = 1

       Ansonsten

              Ergebnis = ggT(a, r)        // Rekursion
```

Pseudocode 1: Der Euklidische Algorithmus als einführendes Beispiel für die Rekursion.

4.2. Suchverfahren

Hartkodierte KIs beruhen auf Algorithmen, um den besten Zug zu finden. Die unkomplizierteste, wohl aber auch die leistungsintensivste Methode zur Ermittlung des besten Spielzuges wäre ein Brute-Force-Suchverfahren. Dabei werden alle möglichen Züge ausprobiert und miteinander verglichen, um den besten Zug zu finden. In der Regel werden alle möglichen Spielausgänge in einem Baumgraphen dargestellt und durchlaufen.

Ein **Algorithmus** ist eine endliche Sequenz von Arbeitsschritten, die sowohl von Menschen als auch von Computern ausgeführt werden können, um ein Problem zu lösen oder eine Berechnung auszuführen.

Eine **Rekursion** ist ein Vorgang, der durch sich selbst definiert ist. Ein Problem auf eine rekursive Art zu lösen bedeutet, es in kleinere Stücke derselben Art zu teilen.

4.2.1. Baumgraph

Im Gegensatz zur Matrixdarstellung ermöglicht der Baumgraph die Darstellung unterschiedlicher Spielzüge in chronologischer Reihenfolge. Im Sinne der Spieltheorie stellt jeder Knoten eine mögliche Spielsituation dar und jede Kante eine Entscheidung.

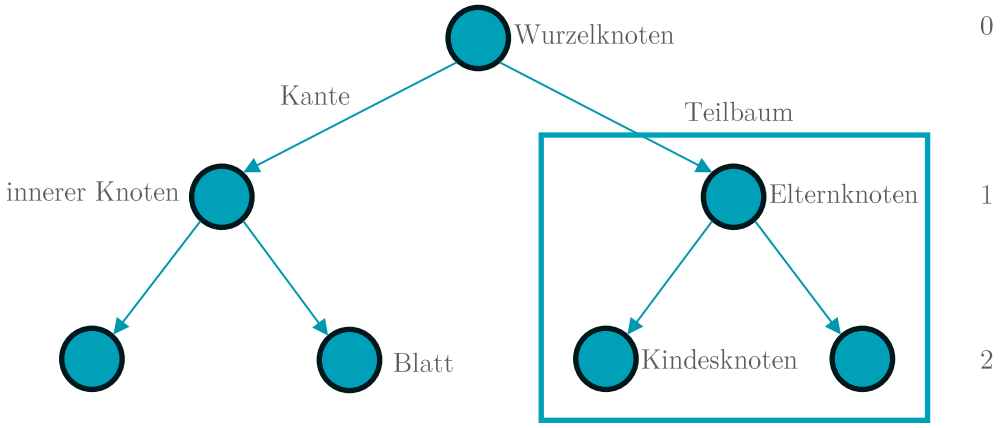


Abbildung 5: Aufbau eines Baumgraphen

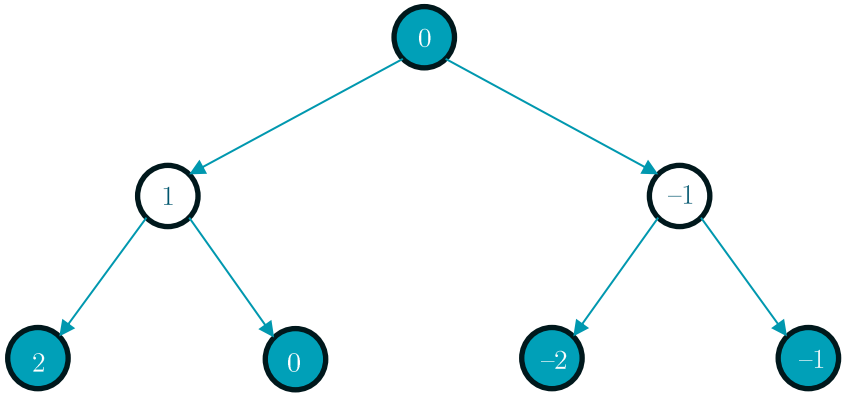


Abbildung 6: Auch im Baumgraphen können verschiedene Spielsituationen unterschiedliche Pay-offs haben.

Angenommen, Amelie und Beatrice spielen gemeinsam Dame. Amelie hat die blauen Steine und Beatrice die Weissen. In diesem Falle könnte die Evaluierung einer Spielsituation so funktionieren: Man zählt die Anzahl der blauen Steine und subtrahiert die Anzahl der weißen Steine. Gibt es mehr blaue als weisse Steine, so ist die Evaluierung positiv. Gibt es auf dem Brett jedoch mehr weisse Steine, so wird die Evaluierung unter null ausfallen. Positive Werte sind folglich vorteilhaft für Amelie, wohingegen Beatrice von negativen Werten profitiert.

Analog stellt in der Abbildung 5 jeder blaue Knoten einen Moment dar, in der Amelie eine Entscheidung treffen muss, wohingegen jeder weisse Knoten einen Entscheidungsmoment für Beatrice darstellt. Da sie ständig untereinander abwechseln, wechseln auch die Knoten abwechslungsweise ihre Farbe.

4.2.2. Der (naive) Minimax-Algorithmus

Der Minimax-Algorithmus durchsucht und bewertet jeden einzelnen Knoten eines Baumes. Das Ziel ist, die gegenseitige beste Antwort zu finden. Ähnlich wie in der Tabelle 7 wird das Nash-Gleichgewicht gefunden, indem der Wert bei allen blauen Knoten (Amelie ist an der Reihe) maximiert wird und bei allen weißen Knoten (Beatrice ist an der Reihe) minimiert wird. Auf diese Weise werden Maxima minimiert und Minima maximiert, bis ein endgültiges Ergebnis, das Nash-Gleichgewicht, gefunden wird.

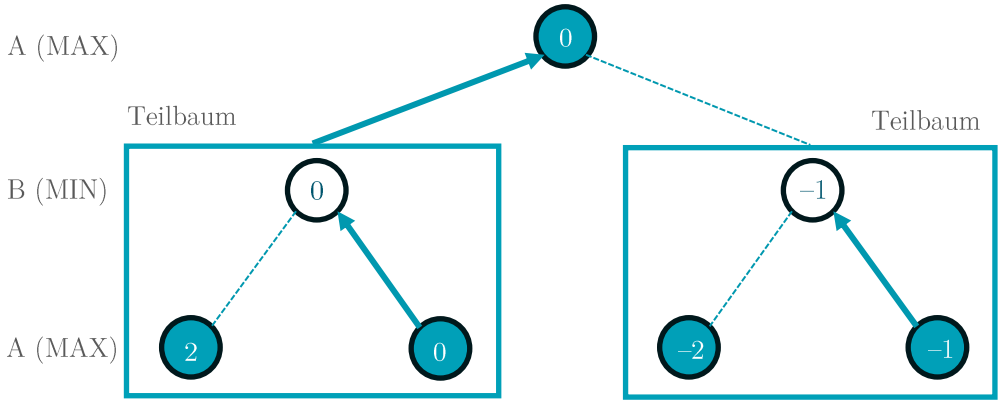


Abbildung 7: Der Minimax-Algorithmus ist rekursiv. Er unterteilt den Baum in all seine Teilbäume und evaluiert sie.

Der Minimax-Algorithmus berechnet den Wert eines Knoten, indem er den Wert seiner Kindesknoten, deren Kindesknoten usw. berechnet. Dies geschieht auf eine rekursive Weise, indem der Algorithmus sich von den Blattknoten zum Wurzelknoten vorarbeitet und die Werte auf dem Weg nach oben trägt, um dem gesamten Spiel einen Wert zuzuordnen. Dieses schrittweise «Zurückbewerten» unterteilt den Baum in immer kleinere Teilbäume, bis eine weitere Unterteilung nicht mehr möglich ist.

Da die Anzahl möglicher Spielsituationen mit jedem Zug exponentiell wächst, kann nicht immer der gesamte Spielbaum durchsucht werden. Daher muss die Suchtiefe begrenzt werden. Dies führt dazu, dass Spielsituationen mit begrenzter Information bewertet werden müssen. Eine heuristische Evaluierungsfunktion bewertet die Knoten nicht objektiv, sondern gibt eine Schätzung ab. Das resultierende Nash-Gleichgewicht ist daher nur eine Schätzung und nicht unbedingt das endgültige Ergebnis.

```
Funktion Minimax(SpielerIn)
        Falls Blatt erreicht dann
                gib die Evaluierung der Spielsituation zurück
        Ansonsten falls SpielerIn = Amelie dann
                Wert = -∞
```

```
        für jeden Kindesknoten
                Wert = max(Wert, Kindesknoten.Minimax(Beatrice))
    Ansonsten falls SpielerIn = Beatrice dann
        Wert = +∞
        für jeden Kindesknoten
                Wert = min(Wert, Kindesknoten.Minimax(Amelie))
    gib den Wert zurück
```

Pseudocode 2: Aufbau des Minimax-Algorithmus

Der **Minimax-Algorithmus** ist ein Verfahren, das einen Spielbaum rekursiv durchläuft. Er evaluiert die Werte der Blätter und trägt sie bis zum Wurzelknoten hinauf. Am Schluss ist somit das Nash-Gleichgewicht gefunden.

4.2.3. Alpha-Beta-Pruning

Der naive Minimax-Algorithmus ist ein Brute-Force-Suchverfahren; er durchsucht alle möglichen Spielzüge, egal ob sie sinnvoll sind oder nicht. Bei grossen Spielbäumen kann dies ein grosser Zeitverlust und eine unnötige Speicherverschwendung darstellen. Integriert man jedoch *Alpha-Beta-Pruning* innerhalb des Minimax-Algorithmus, können grosse Teile eines Baumes weggeschnitten werden, weil das gründliche Durchsuchen an gewissen Stellen zwecklos wäre. Sobald er einen Knoten antrifft, der schlechter ist als ein bisheriger, bricht er mit der Evaluierung dieses Teilbaumes ab.

Das funktioniert dank den zwei Werten Alpha und Beta, welche beim Durchlaufen des Baumes ständig aktualisiert werden. Sie beide stellen die Werte dar, welche die beiden SpielerInnen bei idealer Spielweise erzielen können. Somit kann entschieden werden, ob ein Teilbaum durchlaufen werden soll oder nicht. Am Ende liefert der Alpha-Beta-Algorithmus dasselbe Ergebnis wie der naive Minimax, nur schneller.

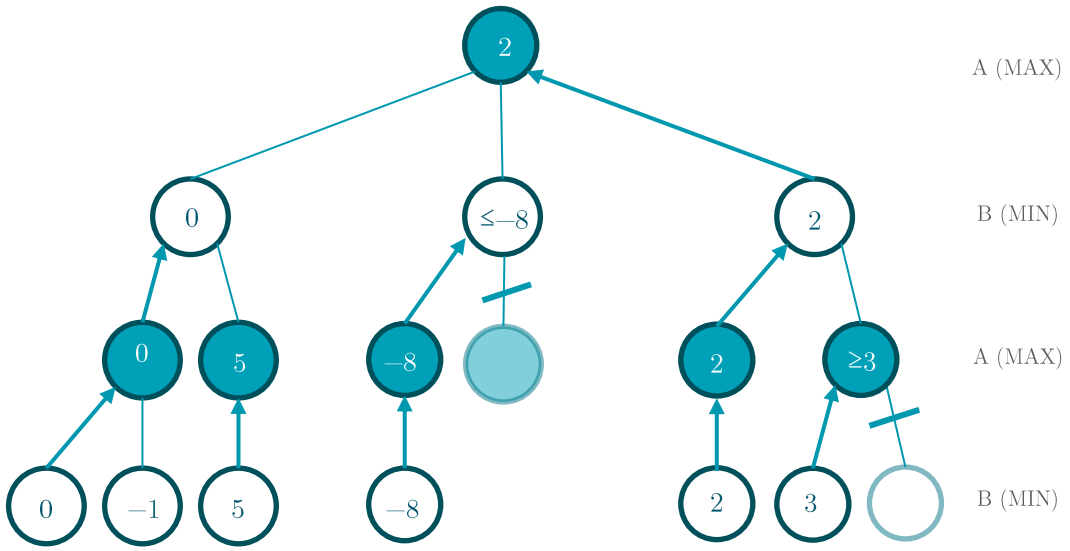


Abbildung 8: Spielbaum mit Alpha und Beta Cut-offs

```
Funktion AlphaBeta(SpielerIn, α, β)
        Falls Blatt erreicht dann
                gib die Evaluierung der Spielsituation zurück
        Ansonsten falls SpielerIn = Amelie dann
                Wert = -∞
                für jeden Kindesknoten
                        Wert = max(Wert, Kindesknoten.Alphabeta(Beatrice, α, β))
                        Falls Wert > β dann
                                abbrechen
                        α = max(Wert, α)
        Ansonsten falls SpielerIn = Beatrice dann
                Wert = +∞
                für jeden Kindesknoten
                        Wert = min(Wert, Kindesknoten.Alphabeta(Amelie, α, β)
                        Falls Wert < α dann
                                abbrechen
                        β = min(Wert, β)
        gib den Wert zurück
```

Pseudocode 3: Der Minimax-Algorithmus mit Alpha-Beta-Pruning

4.3. Multiplayer Suchalgorithmen

In Spielen wie Dame ist der Spielverlauf verhältnismässig überschaubar. Der Baumgraph skizziert alle möglichen Spielausgänge, und ein einfaches Suchverfahren wie der Minimax-Algorithmus kann das Nash-Gleichgewicht finden. Der Minimax ist effektiv, weil Dame von genau zwei Personen gespielt wird. Gewinnen bedeutet, dass die andere Person verlieren wird, so ist es im Interesse beider, sich gegenseitig zu bekämpfen. Sie haben komplementäre Ziele, was das Spiel in diesem Sinne «vollkommen» macht.

Kommen weitere SpielerInnen hinzu, ist die Grenze zwischen Gewinnen und Verlieren nicht mehr binär. Der Wert eines Spiels wird nicht mehr durch eine Zahl, sondern durch eine «Rangliste» repräsentiert. Hier macht es Sinn, für die Pay-offs Vektoren zu verwenden (Luckhardt & Irani, 1986). So können sich SpielerInnen auch mit dem zweiten oder dritten Platz zufriedengeben. Das führt zu einem Problem in Bezug auf die Vorhersehbarkeit des Spiels: Es gibt nicht mehr nur eine rationale Spielweise, und die Ziele der Spieler sind nicht mehr gegensätzlich. Nun können SpielerInnen Allianzen bilden.

4.3.1. Max^n Algorithmus

Der Max^n Algorithmus geht von der optimistischen Annahme aus, dass jede Person für sich spielt. Bei jeder Entscheidung berücksichtigt der Max^n-Algorithmus nur die eigene Punktzahl und nicht die der GegnerInnen.

```
Funktion Max()
        Falls Blatt erreicht dann
                gib die Evaluierung der Spielsituation zurück
        Ansonsten
                Wert = -∞
                für jeden Kindesknoten
                        Wert = max(Wert, Kindesknoten.Max())
        gib den Wert zurück
```

Pseudocode 3: Aufbau des Max^n Algorithmus. Es wird immer der Wert der derzeitigen Person maximiert.

Da der Maxn-Algorithmus immer vom "Best-Case"-Szenario ausgeht, ist er anfällig für schlechte, unvorhergesehene Spielausgänge. Zudem kann kein Alpha-Beta-Pruning durchgeführt werden, da es keine Minima- oder Maxima-Schwellen mehr gibt. (Winands, 2011) Stattdessen können andere Pruning-Verfahren angewendet werden, die zwar eine höhere Suchtiefe erlauben, aber im schlimmsten Fall zu einem schlechten Ergebnis

führen. Es entsteht die Schwierigkeit, die Balance zwischen höherer Suchtiefe und Genauigkeit zu finden. (Shmueli & Zuckerman, 2015)

4.3.2. Paranoia Algorithmus

Der Paranoia-Algorithmus geht von der Annahme aus, dass alle anderen Spieler sich gegen eine Person verschworen haben. Dies führt dazu, dass das Spiel zu einem Zweispieler-Spiel wird, in dem das «Ich» die eigene Punktzahl maximieren will, während «die Anderen» versuchen, diese Punktzahl zu minimieren. Der Paranoia-Algorithmus ist somit ähnlich dem im Pseudocode 3 beschriebenen Minimax-Algorithmus.

Ein Vorteil des Paranoia-Algorithmus ist die Anwendbarkeit des Alpha-Beta-Prunings. Allerdings tendiert er dazu, sich auf das «Worst-Case»-Szenario einzustellen, was dazu führt, dass er äußerst vorsichtig spielt. Es besteht die Gefahr, dass die pessimistischen Prognosen des Paranoia-Algorithmus dazu führen, dass er sich selbst sabotiert. (Winands, 2011)

5. Implementierung

Nun lag es an der Zeit, mein theoretisches Wissen in die Praxis umzusetzen. Für die Umsetzung dieses Projekts habe ich die Programmiersprache C# gelernt, da ich nicht nur den Algorithmus, sondern auch ein Jassprogramm als Begleitung entwickeln wollte.

Ich will jedoch nicht bis in das gründlichste Detail eingehen, sondern eher einen umfassenden Einblick in das Programm geben. Ich werde die wichtigsten Aspekte erklären und auch meine eigenen Überlegungen im Verlaufe dieses Projekts aufzeigen.

5.1. Aufbau des Programms

Objektorientierte Programmiersprachen wie Python, JavaScript oder C# ermöglichen es, innerhalb des Computercodes unsere Welt abzubilden. So können wir ein Player-Objekt definieren, dass eine Hand, eine Ansage oder gestochene Karten besitzt. Den Player-Objekten können auch Fähigkeiten gegeben werden, wie das Spielen oder Stechen von Karten.

Folglich muss die echte Welt abstrahiert werden, um einen virtuellen Jasstisch zu schaffen. Ein Jasstisch besteht üblicherweise aus:

- Einer Trumpffarbe,
- Einem Deck mit 36 Karten,
    - Jede Karte hat einen Zahlenwert (6, 7, 8, 9, 10, J, Q, K, A)
    - Jede Karte hat einen Punktewert und eine Stärke (siehe Kapitel 2)
    - Jede Karte ist entweder Trumpf oder nicht Trumpf
- Vier SpielerInnen
    - Mit je 9 Karten in der Hand
    - Mit einer Ansage
    - Und mit ihren gestochenen Karten
- Einem Stapel, auf denen die Karten gelegt werden.

Am Virtuellen Jasstisch gibt es also neben Player-Objekten auch Karten-Objekte. Diese Elemente befinden sich nicht lose im Programm, sondern werden voneinander getrennt und gruppiert. Diese Mengen werden durch die Listen-Datenstruktur gebildet. Player-Objekte werden aufgelistet {P1; P2; P3; P4}, jedes dieser Player-Objekte besitzt wiederum Handkarten und gestochene Karten {Herz Ass; Schaufel Acht; …}. Die Karten besitzen wiederum Attribute wie Farbe, Punkte- und Stichwert.

Es stellt sich als äussert nützlich heraus, all diese Objekte in einem Container «aufzufangen». Dieser Container werde ich Spielstand-Objekt nennen, da es alle Informationen über den gegenwärtigen Spielstand besitzt: Die Trumpffarbe, die Reihenfolge der Player-Objekte, ihre Ansagen, Karten, etc. Das gesamte Spiel findet innerhalb dieses Objekts statt. Daraus entsteht die Möglichkeit, von den einzelnen Spielsituationen einen Schnappschuss zu machen, um diesen zu kopieren oder zu einem späteren Zeitpunkt zu importieren.

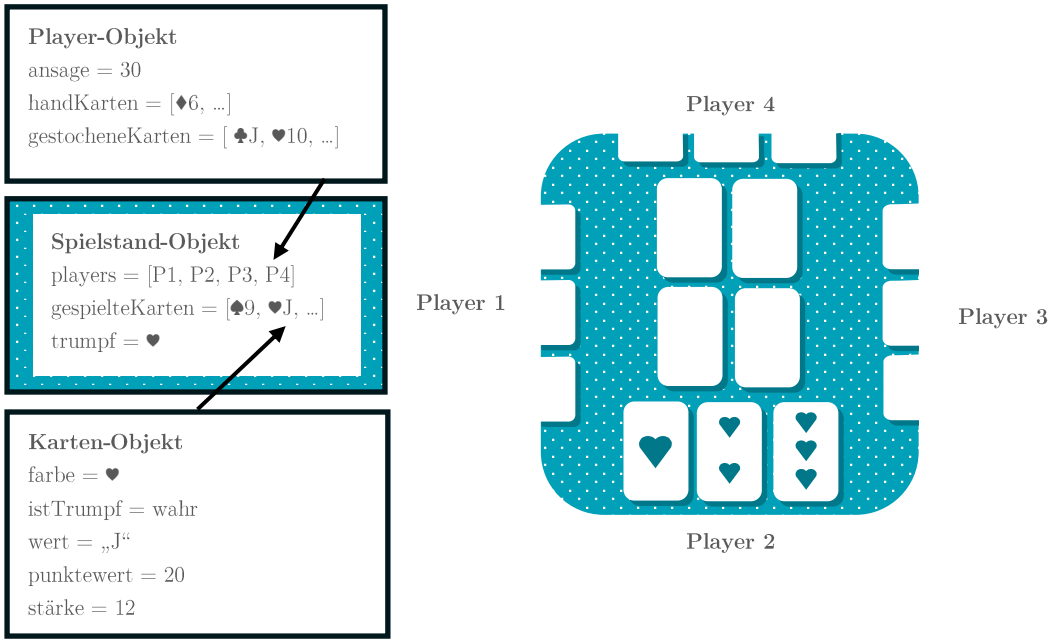


Abbildung 9: Aufbauskizze des Programms

5.1.1. Grundlegende Funktionen

Der Verlauf eines Kartenspiels geschieht in der echten Welt fast von selbst, aber innerhalb eines Programms muss jeder einzelne Schritt erklärt und durchgeführt werden. Das gesamte Spiel wird durch unterschiedliche *Funktionen* am Computer vermittelt. Die wichtigsten Funktionen sind:

- Erstellen, Mischen und Verteilen eines Kartensets
- Spielbarkeit von Karten überprüfen
- Herausfinden, welcher Player die stärkste Karte gespielt hat
- Stechen
- Zählen von Punkten
- Erstellen einer Rangliste
- etc.

Ausserdem braucht es eine Spielschlaufe, welche die oben genannten Funktionen wiederholt anwendet, bis das Spiel zu Ende geht.

```
Funktion Spielschlaufe
        wähle Zufällig eine Trumpffarbe
        generiere ein Kartendeck von 36 Karten
        mische die Karten
        verteile die Karten an die SpielerInnen
        lass die SpielerInnen ihre Ansage machen
        für neun Runden
                à vier SpielerInnen
                        herausfinden, welche Karten gespielt werden dürfen
                        spielt eine Karte
                wenn vierte Karte gespielt wurde, dann
                        stechen
        wenn alle Karten gespielt, dann
                erstelle Rangliste


Funktion stechen
        herausfinden, welches die stärkste Karte ist
        herausfinden, von wem die stärkste Karte gespielt wurde
        gib die gestochenen Karten an die Person, welche den Stich gewonnen
        hat
        ändere die Reihenfolge der SpielerInnen


Funktion Rangliste
        ermittle die Differenzen aller SpielerInnen
        sortiere die SpielerInnen nach absteigender Differenz
        stelle die Rangliste tabellarisch dar
```

Pseudocode 4: Spielschlaufe des Differenzlers und einige der Funktionen

Die Spielbarkeit von Karten habe ich bereits im Kapitel 2.2.1 (Abbildung 1) besprochen. Ich musste dieselbe Logik nur noch innerhalb meines Programms umsetzen. Ich bezweifle jedoch, dass eine ausführliche Erklärung der einzelnen Funktionen zum Verständnis dieser Arbeit beitragen würde. Der Computercode ist einzig eine formale Schreibweise für Prozesse, die für uns Menschen gegeben sind. Folglich scheint mir eine Erklärung für das

Mischen oder Verteilen von Karten sehr banal. Deshalb werde ich nicht genauer auf die technischen Aspekte dieser Funktionen eingehen.

5.2. Erste Phase: «offener» Differenzler

Wie ich es bereits im Kapitel 4.3 beschrieben habe, stellt schon allein die Einführung von mehreren SpielerInnen eine grosse Herausforderung bezüglich der Vorhersehbarkeit dar. Zugleich ist der Differenzler ein Spiel mit imperfekter Information. Laut Zermelo sind somit nur zwei von fünf Bedingungen für ein vorhersehbares Spiel erfüllt (Zermelo, 1913):

- ✗ **Das Spiel wird von zwei Personen gespielt.**
- ✗ **Es ist ein Nullsummenspiel.**
- ✓ **Das Spiel ist endlich.**
- ✗ **Das Spiel hat perfekte Information.**
- ✓ **Es gibt keine zufälligen Einflüsse[4].**

Der Differenzler ist zwar mathematisch unvorhersehbar, doch ich wollte herausfinden, wieviel man mit einem hartkodierten Suchverfahren erreichen konnte. Dafür musste ich mich langsam an den Differenzler heranarbeiten. Ich begann mit der Vereinfachung der Spielsituation und erschuf den «offenen» Differenzler.

Im offenen Differenzler wissen alle SpielerInnen die Karten ihrer GegnerInnen. Das Spiel hat folglich perfekte Information. Da ich bereits das Spielstand-Objekt definiert hatte, konnte ich Problemlos auf die darin beinhalteten Informationen zugreifen. Punkt 1 und 2 konnte ich ebenfalls vernachlässigen, da bei Multiplayer-Spielen mit perfekter Information die zuvor beschriebenen Algorithmen Max^n und *Paranoia* anwendbar sind.

Demnach war mein erstes Ziel, den offenen Differenzler zu meistern, um später mein Verfahren für den «verdeckten» Differenzler anwendbar zu machen. Würde sich jedoch herausstellen, dass ein Suchverfahren für den offenen Differenzler unmöglich ist, so könnte ich bereits dort abbrechen.

5.2.1. Teil I: Ansage

Die Evaluierung eines gegebenen Spielsituation soll aussagen, wie gut oder schlecht ein bestimmter Zug ist. Im Kontext des Differenzlers ist das die Diskrepanz zwischen angesagter und erreichter Punktzahl. Ohne Ansage ist keine Evaluierung des Spielstandes möglich, und demnach ohne Evaluierung auch kein Suchverfahren. So war die erste Frage, mit der ich mich beschäftigt habe:

[4] Die Karten sind zwar zufällig verteilt, im weiteren Verlauf des Spiels gibt es aber keine weiteren zufälligen Einflüsse.

«Wie kann ich anhand der Karten in meiner Hand herausfinden, wie viele Punkte ich Ansagen muss?»

Es wird sich später herausstellen, dass diese Frage nicht nur am Anfang des Spiels auftaucht, sondern uns in ähnlicher Form während dem gesamten Prozess begleitet.

Das schwierige dabei ist, dass es bisher keine Zauberformel gibt, welche die perfekte Ansage ermitteln kann. Stattdessen gibt es nur eine grobe Faustregel, die sich durchgesetzt hat: Die Ansage ist der doppelte Punktewert jeder Trumpfkarte und für jedes weitere Ass je 11 Punkte. (Muff, 2022b)

Davon ausgehend, dass alle SpielerInnen diese selbe Strategie anwenden, wird die totale Ansage immer 157 Punkte sein. Es wird folglich nie zu wenig oder zu viel angesagt. Die resultierende Ansage ist jedoch nur ein Richtwert, und berücksichtigt weder die Karten noch die Spielweise der anderen SpielerInnen.

Um eine Ansage zu machen, die mehr auf die jeweilige Situation zugeschnitten ist, habe ich am Anfang des Spiels etwa 1000 rein Zufällige Spiele durchsimulieren lassen, um somit die durchschnittliche Punktzahl zu ermitteln. Auch hier war die totale Ansage immer 157 Punkte, aber es blieb unklar, ob diese Methode viel besser war als die bewährte Faustregel.

5.2.2. Teil II: Spielbaum

Zwar habe ich bereits im Kapitel 4.2.1 beschrieben, wie Baumgraphen aufgebaut sind, doch wie werden solche riesigen Gebilde überhaupt erstellt?

Für den Anfang hilft die Vorstellung, wie man einen Spielbaum von Hand machen würde: Man nimmt der jetzige Spielstand als Anfangsknoten, und zeichnet von dort aus für jeden möglichen Spielzug eine Abzweigung. Dann wandert man zur ersten Abzweigung, und macht für jede der möglichen Spielzüge weitere Abzweigungen. Diesen Prozess würde man so oft wiederholen, bis der ganze Spielbaum aufgezeichnet ist. Ähnlich geht es auch innerhalb des Jassprogramms. Dort habe ich ein sogenanntes Knotenobjekt eingeführt:

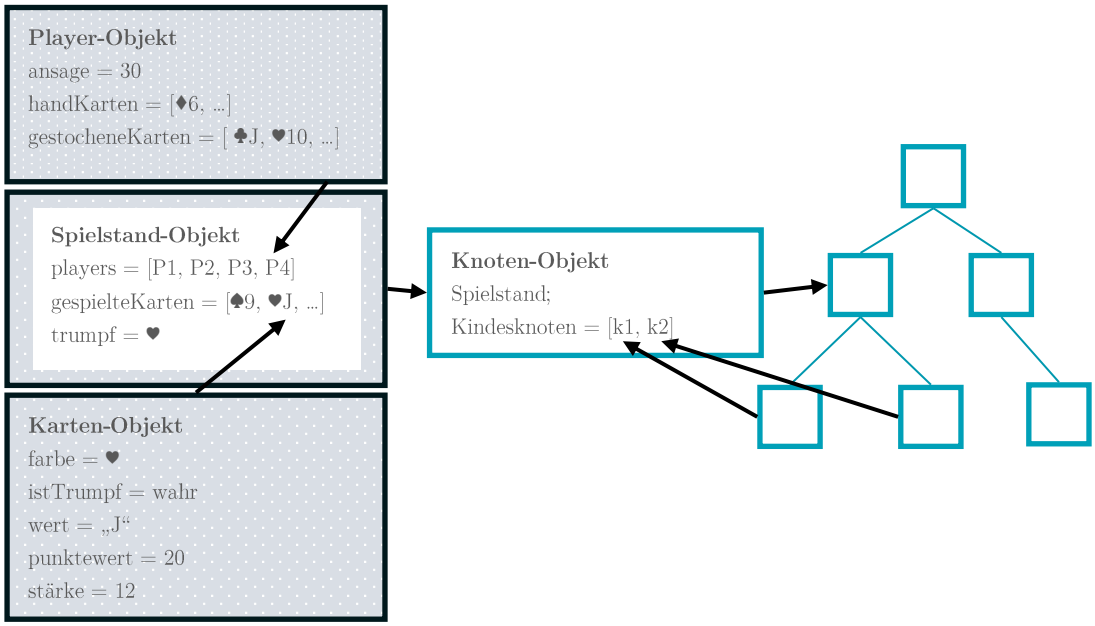


Abbildung 10: Aufbauskizze des Spielbaumes

Jeder Knoten besteht aus einem einzigartigen Spielstand-Objekt und steht in Verbindung zu seinen Eltern- und Kindesknoten. Alle diese Verbindungen bilden ein grosses Ganzes, einen Spielbaum.

Zum Erstellen eines Spielbaumes müssen wir also systematisch alle möglichen Knoten bilden und sie mit ihren jeweiligen Kindesknoten verbinden. Die Systematik entsteht durch das rekursive Verfahren, das ich zuvor definiert habe: Man nehme einen Knoten und bildet für jeden Spielzug einen Kindesknoten. Für jeden dieser Kindesknoten wiederholt man dasselbe Verfahren, bis der ganze Spielbaum aufgezeichnet ist.

```
Knoten(Spielstand, Tiefe) // erstellt einen Knoten
        Lege den Spielstand als Attribut des Knotens fest
        Finde spielbare Karten
        Falls Tiefe = 0 oder keine Spielbare Karten dann
                Knoten ist ein Blatt
        Ansonsten
                Knoten ist kein Blatt
                für jede spielbare Karte
                        kopiere den Spielstand
                        Spiele die Karte innerhalb des kopierten Spielstandes
                        Falls 4 gespielte Karten dann
                                Stechen
```

```
                    Füge Knoten(kopieter Spielstand, Tiefe-1) zu den
                    Kindesknoten hinzu  // Rekursion
```

Pseudocode 5: Funktion zum Erstellen eines Spielbaumes

Für das Erstellen des Spielbaumes braucht es demnach zwei Parameter: der jetzige Spielstand und die Suchtiefe. Am Anfang legen wir den Spielstand des Knotens fest. Danach überprüfen wir, wer an der Reihe ist, und welche Karten diese Person zu Verfügung hat. Für jede mögliche Karte erstellen wir eine Kopie des Spielstand-Objekts und spielen die jeweilige Karte, sodass quasi «Parallelwelten» entstehen. Für jede dieser «Parallelwelten», wird ein Kindesknoten erstellt, und das kopierte Spielstand-Objekt wird als Spielstand des Kindesknotens festgelegt. Nun wiederholen wir denselben Prozess für die Kindesknoten, wobei die Suchtiefe um eins vermindert werden muss, sodass der rekursive Prozess irgendwann zu Ende geht.

Da der offene Differenzler ein Spiel mit perfekter Information ist, macht es keinen Sinn, den Spielbaum ständig neu zu bilden. Stattdessen kann man aus dem Spielbaum einen Knoten wählen, und diesen als neuen Wurzelknoten definieren. Ausserdem kann durch eine *Update*-Funktion am bisherigen Spielbaum angeknüpft werden.

```
Funktion Update(Tiefe) // Erweiert einen Spielbaum

       Falls Tiefe > 0

              Falls Knoten ist Blatt dann

                     // Erweitere den Baum

                     Knoten = Knoten(Spielstand, Tiefe)

                     Knoten ist kein Blatt mehr

              Ansonsten

                     // Durchlaufe den Baum, bis Blatt erreicht

                     Für jeden Kindesknoten

                            Kindesknoten.Update(Tiefe)
```

Pseudocode 6: Die Update-Funktion erweitert einen Spielbaum

Diesen Prozess, den ich hier beschrieben habe, nennt sich *Depth-First Search*, oder auch Tiefensuche, da der Baum gebildet wird, indem er bei der ersten Abzweigung anfängt und sich bis zu den untersten Blättern hinunterarbeitet, und erst nachher die anderen Abzweigungen erforscht. Somit muss die Bildung des Baumes nicht zeitlich, sondern mithilfe einer Tiefe begrenz werden.

Dank dieser Struktur können die Spielbäume mit den Max^n- und dem Paranoia-Algorithmus durchlaufen werden, wie ich sie bereits im Kapitel 4.3.1 und 4.3.2 beschrieben habe. Das Einzige, was noch definiert werden musste, war die *Evaluierungsfunktion*.

5.2.3. Teil III: Heuristische Evaluierung

Der Zweck einer Evaluierungsfunktion ist, einen gegebenen Spielstand zu bewerten. Er bildet das Herzstück des Suchalgorithmus, denn wenn die Bewertung falsch ist, so gibt auch der Suchalgorithmus ein unbrauchbares Resultat zurück.

Am Anfang sah die Evaluierungsfunktion eines Blattknotens für alle Player-Objekte so aus:

$$\textit{Differenz} = \textit{angesagte Punktzahl} - \textit{gestochene Punkte}$$

Auf den ersten Blick scheint diese Formel logisch, und der jeweilige Suchalgorithmus müsste gezwungenermassen den Spielzug finden, welcher die tiefste Differenz garantiert. Sie ist jedoch unzureichend, da aufgrund des kombinatorischen Wachstums der Spielzüge nur eine begrenzte Suchtiefe möglich ist. Das hat zur Folge, dass die Evaluierung unvollständig ist. Das erkennt man am folgenden Beispiel:

Der Suchalgorithmus hat fast den untersten Knoten erreicht und evaluiert nun den dazugehörigen Spielstand. Ein Player hat zwei Karten zur Auswahl: Ein Nell mit einem Wert von 14 Punkten, und eine Sechs mit einem Wert von null Punkten. Da zuvor ein Trumpf-Bauer gespielt wurde, ist es unmöglich, den Stich für sich zu gewinnen. Sowohl das Nell als auch die Sechs werden null Punkte einbringen. Aus diesem Grund werden die beiden Spielzüge als Gleichwertig betrachtet. Es wird nicht berücksichtigt, dass im nächsten Zug das Nell ein Bock sein wird und garantiert mindestens 14 Punkte einbringen kann.

Diese Unvollständigkeit der Evaluierungsfunktion ist offensichtlich gravierend, wenn die schlechteste Karte im Spiel gleich gewertet wird wie die zweitbeste. Demnach darf die Evaluierung nicht nur die angesagte Punktzahl und die gestochenen Punkte berücksichtigen, da es vor allem in den ersten Zügen des Spiels noch sehr hohe Differenzen gibt. Es müssen zusätzlich noch die Karten berücksichtigt werden, die sich in den Händen der SpielerInnen befinden:

$$\textit{Differenz} = \textit{angesagte Punktzahl} - (\textit{gestochene Punkte} + \textit{prognostizierte Punkte})$$

Dabei stellt sich eine ähnliche Frage wie im Kapitel 5.2.1:

«Wie kann ich anhand der Karten in meiner Hand herausfinden, wie viele Punkte ich machen werde?

Somit sind wir wieder am selben Punkt angelangt, wie am Anfang. Klar ist: Es gibt keine absolut robuste Formel, die prognostizieren kann, wie viele Punkte man mit einer gegebenen Hand machen wird. Auch hier könnte man die Faustregel anwenden, wie ich sie zuvor beschrieben habe, aber dadurch würde das Problem bestehen bleiben, dass sich diese Prognose weder die Karten der anderen SpielerInnen noch die Böcke berücksichtigt.

Um dieses Problem zu umgehen, entwickelte ich eine andere Variante:

Ich überprüfte alle möglichen Kombinationen von vier Karten (das wären höchstens $9^4 \times 4$ Möglichkeiten) und rechnete für jede Karte die Summe aller gestochenen Punkte geteilt durch die Anzahl Kombinationen, welche diese Karte beinhalten (das wären höchstens $9^3 \times 4$ Möglichkeiten). Auf diese Art konnte ich den durchschnittlichen Stichwert ermitteln. Schwache Karten wie die Sechs bekamen somit sehr tiefe Prognosen, da ein Stich nur unter spezifischen Umständen möglich war, wohingegen stärkere Karten oder Böcke weitaus höhere Prognosen erhielten.

Um eine Prognose zu erstellen, wurde der durchschnittliche Stichwert aller Karten addiert. Die Prognostizierte Punktzahl und die bereits gestochenen Punkte wurden dann von der Ansage subtrahiert, das Resultat war die Evaluierung des Spielstandes.

5.2.4. Teil IV: Suchalgorithmen

Nun war dank der heuristischen Evaluation die Basis für die Suchalgorithmen gegeben. Ich konnte die Algorithmen problemlos in das Programm implementieren, wobei der Max^n die Differenzen nicht mehr maximieren, sondern minimieren musste. Ich stellte mir dabei aber die Frage, ob das wirklich der Sinn und der Zweck des Differenzlers ist.

Zwar ist es von Vorteil, die eigene Differenz so tief wie möglich zu halten, aber es kommt nicht nur auf die eigene Differenz an, sondern auch auf die der anderen. Wenn die tiefstmögliche Differenz nur den zweiten Platz garantieren kann, so wird man vielleicht auf die Idee kommen, die Differenzen der anderen SpielerInnen zu sabotieren. So realisierte ich, dass es für die einzelnen SpielerInnen nicht unbedingt von Interesse ist, die Differenzen insgesamt tief zu halten. Es ist viel nützlicher, eine tiefere Differenz zu besitzen als alle anderen. Zwar ist dieses Verhalten im normalen Jass verpönt (Muff, 2022a), ich wollte aber dennoch diese Strategie ausprobieren und erweiterte die bisherigen Algorithmen.

Neben den üblichen Algorithmen erstellte ich eine geordnete Version. Diese bezieht sich nicht primär auf die Differenzen, sondern auf die Platzierung innerhalb der Rangliste. Erst als zweites Kriterium wird dann die eigentliche Differenz berücksichtigt. Ich musste somit die Evaluierungen, die Minimierungsfunktion *Min()* und die Maximierungsfunktion *Max()* neu definieren. Die Maximierungsfunktion vergleicht dabei immer nur zwei Evaluierungen und gibt die grössere der beiden zurück. Die Minimierungsfunktion ist

dadurch definiert, dass sie das Gegenteil der Maximierungsfunktion als Resultat zurückgibt.

Für die Evaluierung habe ich ein Evaluierungsobjekt eingeführt. Es beinhaltet eine Rangliste, die Differenzen aller SpielerInnen und ob diese Differenzen positiv oder negativ sind. Die Idee dabei ist, dass man lieber 2 Punkte zu wenig hat als 2 Punkte zu viel.

```
Evaluation Max(Evaluation x, Evaluation y, Player, max)
       Falls max = wahr // Maximiere
              Falls Differenz des Player bei x > Differenz bei y, dann
                     gib Evaluation x als Resultat zurück
              Ansonsten falls beide Differenzen gleich, dann
                     Falls x eine positive Differenz ist, dann
                            gib Evaluation x als Resultat zurück
              Ansonsten
                     gib Evaluation y als Resultat zurück
       Ansonsten // Minimiere
              Falls x = Max(x, y, Player, wahr), dann
                     gib Evaluation y als Resultat zurück
              Ansonsten
                     Gib Evaluation y als Resultat zurück


Evaluation geordneterMax(Evaluation x, Evaluation y, Player)
       Falls = wahr // Maximiere
              Falls Rang des Players bei x > Rang bei y, dann
                     gib Evaluation x als Resultat zurück
              Ansonsten falls beide Ränge gleich, dann
                     gib Max(x, y, Player) als Resultat zurück
              Ansonsten
                     gib Evaluation y als Resultat zurück
       Ansonsten // Minimiere
              Falls x = geordneterMax(x, y, Player, wahr), dann
                     gib Evaluation y als Resultat zurück
              Ansonsten
```

Pseudocode 7: Evalueriungsfunktionen Max() und geordneterMax()

5.2.5. Zusatz: Dynamische Tiefe

Ein Problem, das mir währenddessen aufgefallen ist, ist die limitiere Suchtiefe. Dabei war es sehr schwierig abzuschätzen, wie tief höchstens gesucht werden darf, damit in einem realistischen Zeitfenster ein Resultat gefunden wird. Ich wusste jedoch, dass im Verlauf des Spiels die Anzahl der möglichen Entscheidungen abnimmt, und dass ab einer gewissen Stelle das Spiel komplett vorhersehbar sein muss.

Um diesen Aspekt zu erforschen, habe ich eine Million zufällige Spiele simuliert, und an jeder Stelle gezählt, wie viele Möglichkeiten zur Auswahl standen. Diese Werte habe ich dann tabellarisch festgehalten:

	1	2	3	4	5	6	7	8	9	Ø
1	0	0	0	0	0	0	0	0	1000000	9
3	59450	122444	177252	224918	199857	106656	31717	4729	72977	4.291631
3	111031	224417	243071	182964	105730	44757	12313	1835	73882	3.583935
4	145025	275791	257845	153291	66681	21788	5227	762	73590	3.252434
5	0	0	0	0	0	0	0	1000000	0	8
6	55883	127721	210189	236225	164280	66569	14470	124663	0	4.2062
7	116299	219852	243101	178524	89506	29279	5579	117860	0	3.604539
8	159909	265906	244622	143299	57220	15734	2757	110553	0	3.28301
9	0	0	0	0	0	0	1000000	0	0	7
10	87463	191223	253575	197483	89105	21904	159247	0	0	3.712244
11	133399	236361	244671	156686	61787	13653	153443	0	0	3.431832
12	165792	265196	237263	130632	45488	9523	146106	0	0	3.237821
13	0	0	0	0	0	1000000	0	0	0	6
14	135337	243300	239131	132557	40591	209084	0	0	0	3.327017
15	167574	262193	226102	111454	30152	202525	0	0	0	3.181992
16	188231	275383	220056	99226	24941	192163	0	0	0	3.073752
17	0	0	0	0	1000000	0	0	0	0	5
18	199882	272122	190350	69854	267792	0	0	0	0	2.933552
19	214160	274242	183306	63609	264683	0	0	0	0	2.890413
20	223918	277779	179926	60568	257809	0	0	0	0	2.850571
21	0	0	0	1000000	0	0	0	0	0	4
22	256357	262935	128974	351734	0	0	0	0	0	2.576085
23	263982	261539	124549	349930	0	0	0	0	0	2.560427
24	267974	262507	124329	345190	0	0	0	0	0	2.546735
25	0	0	1000000	0	0	0	0	0	0	3
26	299181	219606	481213	0	0	0	0	0	0	2.182032
27	301243	217221	481536	0	0	0	0	0	0	2.180293
28	303238	219218	477544	0	0	0	0	0	0	2.174306
29	0	1000000	0	0	0	0	0	0	0	2
30	311983	688017	0	0	0	0	0	0	0	1.688017
31	312449	687551	0	0	0	0	0	0	0	1.687551
32	314133	685867	0	0	0	0	0	0	0	1.685867
33	1000000	0	0	0	0	0	0	0	0	1
34	1000000	0	0	0	0	0	0	0	0	1
35	1000000	0	0	0	0	0	0	0	0	1
36	1000000	0	0	0	0	0	0	0	0	1
										2.4462E+16

Tabelle 8: Die Anzahl Zugmöglichkeiten und ihre Häufigkeit. Die y-Achse (0-36) beschreibt die Position innerhalb des Spiels, die x-Achse beschreibt, wie viele Zugmöglichkeiten es in der Position y gibt.

Ich konnte abschätzen, wie viele Blattknoten ab einer gewissen Tiefe entstehen würden, und somit die Suchtiefe abgrenzen. Dies ermöglichte es mir, gegen den Schluss Suchtiefen

von bis zu 15 zu erreichen, und so das Spiel ab etwa der Hälfte vorhersehbar zu machen. Folgende Formel gab eine grobe Abschätzung, wobei k den derzeitigen Zug darstellt.

$$Anzahl\ Blattknoten \approx \prod_{k}^{36} \emptyset_k$$

5.2.6. Zwischenresultate

Ich merkte mit der Zeit, dass meine erste Implementierung alles andere als Ideal war. Einerseits beanspruchte der Spielbaum die gesamte Kurzzeitspeicherkapazität, da mit jedem Knoten ein weiterer Spielstand kopiert wurde, der wiederum 36 Karten beinhaltete. Das hatte zur Folge, dass der gesamte Kurzzeitspeicher von Millionen und Milliarden von Kartenobjekten beansprucht wurde. Andere Informationen wie Ansagen oder Trumpffarben wurden ebenfalls millionenfach kopiert, obwohl diese Werte während dem ganzen Spielverlauf konstant bleiben.

Ein anderes Problem war die unzuverlässige Bestimmung der Suchtiefe. Mit meinem Suchverfahren, der Tiefensuche, gibt es keine Möglichkeit, die Suchtiefe zeitlich zu begrenzen und ein nützliches Ergebnis innerhalb eines vernünftigen Zeitraumes zu garantieren. Beim Paranoia-Algorithmus konnten grosse Teile des Baumes abgeschnitten werden, was beim Max^n keine Option war. Das hatte gravierende Auswirkungen auf die Laufzeit.

Auch war es ineffizient, in einem Schritt den Spielbaum zu kreieren und ihn erst in einem zweiten Schritt zu evaluieren, was zur Folge hatte, dass der Spielbaum unnötigerweise zweimal durchlaufen werden musste.

Die offensichtlichste Schwierigkeit war jedoch die heuristische Evaluierung der Spielsituation, da ich aufgrund der unzähligen Parameter wie Zufall, Spielweise oder Ansage nicht beurteilen konnte, ob meine Evaluierungsfunktion gut war. Die Resultate sahen jedoch nicht schlecht aus – die Differenzen waren meistens innerhalb eines vernünftigen Bereiches, was mich dazu anspornte, zum verdeckten Differenzler überzugehen.

	Durchschnitt	Median	Varianz	Standartabweichung
Paranoia	15.67	13	147.77	12.15
Max^n	16.11	13	163.39	12.78
Paranoia geordnet	17.41	14	187.13	13.67
Max^n geordnet	16.90	14	184.65	13.58

Tabelle 9: Differenzen der unterschiedlichen Suchalgorithmen

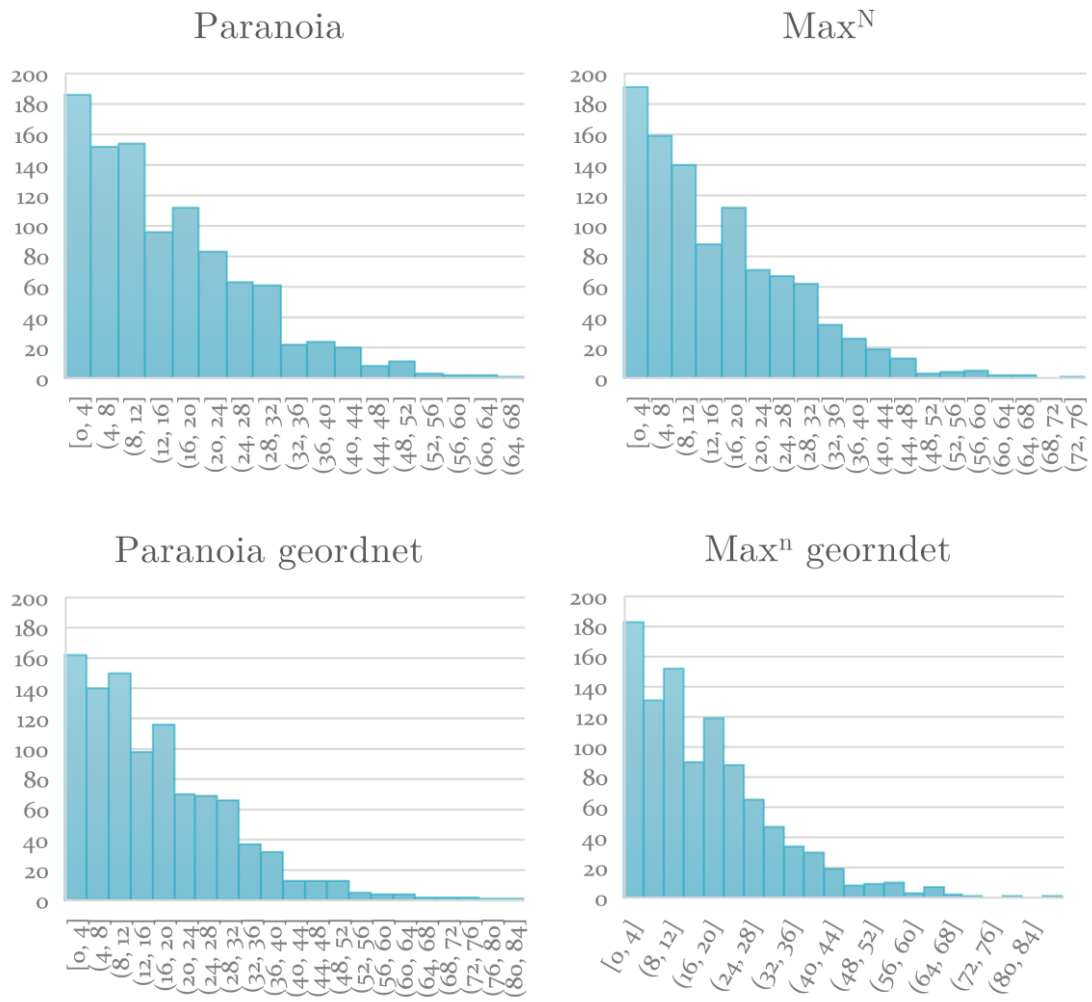


Tabelle 10: Verteilung der unterschiedlichen Differenzen bei 1000 Spielen

5.3. Zweite Phase: «Verdeckter» Differenzler

Ich hatte nun ein Programm, das anhand eines Spielbaumes und einem Suchalgorithmus einen Spielzug auswählt, den es für den besten hält. Während diesen ganzen Prozess ging ich von der Annahme aus, dass die Spielsituation für alle SpielerInnen bekannt ist. Nur so konnte ich den Max^n- und den Paranoia-Algorithmus anwenden. Beim verdeckten, beziehungsweise normalen Differenzler trifft diese Annahme nicht mehr zu.

Ich überlegte mir, dass ich abgesehen von einigen Optimierungen den Algorithmus in seiner Grundstruktur unverändert lasse. Das bedeutet aber, dass ich eine unbekannte Spielsituation so umformen muss, damit der Algorithmus anwendbar ist.

5.3.1. Exkurs I: Der Monte-Carlo-Algorithmus

Im Gegensatz zu den Algorithmen, die ich bisher erwähnt habe, hat der Monte-Carlo Algorithmus keine klar festgelegte Struktur. Er ist dadurch definiert, dass er Resultate mithilfe von zufälligen Proben oder Simulationen produziert.

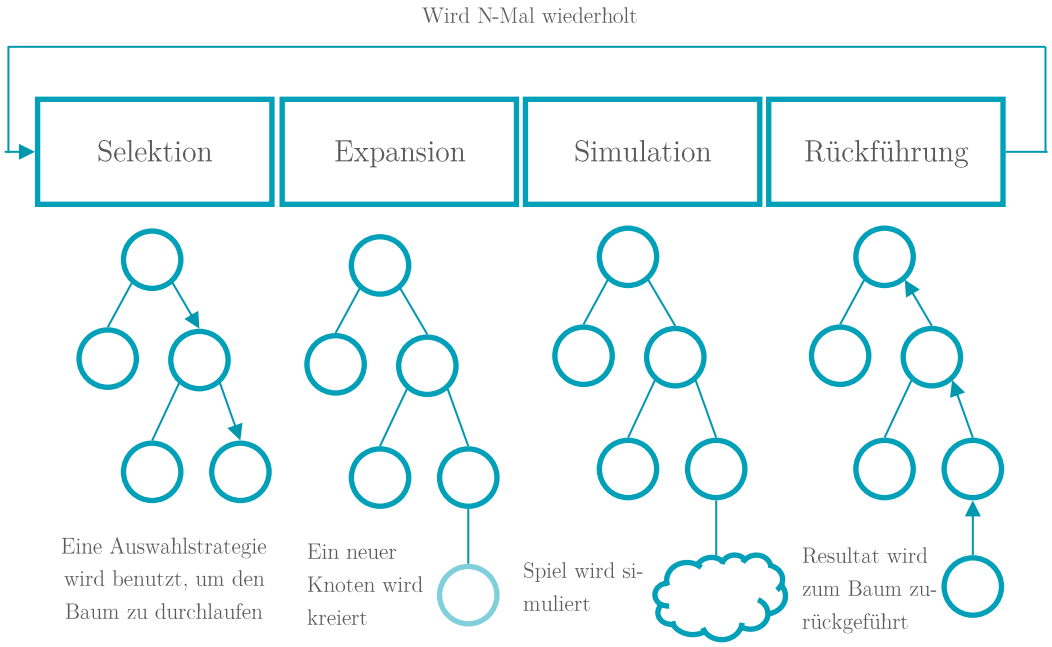


Abbildung 11: Aufbau des Monte-Carlo Algorithmus (Nijssen, 2013)

Der Monte-Carlo Algorithmus bildet schrittweise einen Spielbaum, indem er gewisse Züge nach Kriterien wie Gewinnrate oder durchschnittlichen Punkte auswählt, und diese am Suchbaum anfügt. Danach erforscht er diesen Knoten, indem er einige Spiele simuliert, und das Resultat als Evaluierung an den Spielbaum zurücksendet.

5.3.2. Umgang mit Unwissen

Als ich über die mögliche Implementierung eines Monte-Carlo-Algorithmus nachdachte, begann ich, an seiner Effektivität für den Differenzler zu zweifeln. Ich betrachtete dafür insbesondere die Anfangssituation des Spiels: Bevor eine Karte gelegt wird, sind einzig die eigenen Karten bekannt. Das bedeutet, dass von allen $\binom{27}{9} \times \binom{18}{9}$ möglichen Verteilungen von 27 Karten an 3 SpielerInnen jede die gleiche Wahrscheinlichkeit besitzt. Aus diesem Grund macht es besonders in dieser Phase des Spiels nur wenig Sinn, Wahrscheinlichkeiten zuzuteilen oder mögliche Spielsituationen zu simulieren.

Ich merkte jedoch, dass im Verlauf des Spiels Informationen gewonnen werden, zum einen durch die Karten, die gespielt werden, und solche, die nicht gespielt werden. Durch Ausschlussverfahren war es möglich herauszufinden, ob eine Person eine gewisse Farbe besitzt oder nicht. Sagen wir zum Beispiel, Herz ist Trumpf und die ausgespielte Farbe

ist Schaufel. Würde eine Person nun Kreuz Spielen, so ist es klar, dass sie keine Schaufel besitzen kann.

Ich stellte mir dabei eine Art Tabellenverzeichnis vor, in der jede Karte einen bestimmten Standort besitzt. Dieser Standort wird mit jedem Spielzug aktualisiert, sodass ab einer gewissen Spielzeit ein umfassendes Wissen über die Standorte der Karten besteht.

	P1	P12	P3	P4
♥ A	0.33	0.33	0.33	0
♦ J	0	0.5	0.5	0
♣ 6	0	0	0	1
...	...	...	...	...

Tabelle 11: Verzeichnis für die Standorte der jeweiligen Karten

Je weiter das Spiel fortgeschritten ist, desto besser kann man das Spielstand modellieren und evaluieren.

Mein Ziel war es, anhand dieses Tabellenverzeichnis eine mögliche Spielsituation zu bilden, um innerhalb dieser Situation die beste Antwort zu finden. Ich war mir bewusst, dass mit hoher Wahrscheinlichkeit nicht der wahre Spielstand gebildet wird. Diese Tatsache ist und bleibt unausweichlich. Ich sah keinen Vorteil darin, alle eventuellen Kartenkonstellationen zu berücksichtigen, da ein besseres Resultat nicht garantiert war. Lieber würde ich mich nur auf *eine* wahrscheinliche Spielsituation beziehen, welche im Verlauf des Spiels genauer wird.

5.3.3. Exkurs II: Wie Sudokus gelöst werden

Per Zufall stiess ich auf eine Methode, die mir helfen könnte, einen Spielstand zu modellieren: Der *Wave Function Collapse Algotirhmus* (WFCA). Angewendet wird er normalerweise für die prozedurale Generierung von Bildern. Der WFCA löst dabei ein «Puzzle», dass gewisse Regeln beinhaltet. Diese Regeln beziehen sich immer auf Zellen, ähnlich wie in einem Sudoku-Gitter. Dort gilt: In jeder Spalte und Zeile kommt jede Zahl nur einmal vor, dasselbe gilt auch für die markierten 3×3 Felder. (Donald, 2020)

Innerhalb meines Tabellenverzeichnis würden die Regeln so aussehen: Jede Person muss 9 Karten besitzen, das bedeutet, dass pro Spalte die Summe aller Wahrscheinlichkeiten 9 betragen muss. Da jede Karte nur einer Person gehören kann, muss die Summe aller Wahrscheinlichkeiten pro Zeile immer 1 betragen.

Oft ist es der Fall, dass eine Zelle keinen klar definierten Zustand besitzt, das bedeutet innerhalb eines Sudokus, dass in einer Zelle mehrere Zahlen möglich sind. Innerhalb meines Tabellenverzeichnis zeigt sich diese Ungewissheit bei Zellen, die nicht klar mit 0 oder 1 versehen sind. Diesen Zustand würde man als *Superposition* bezeichnen.

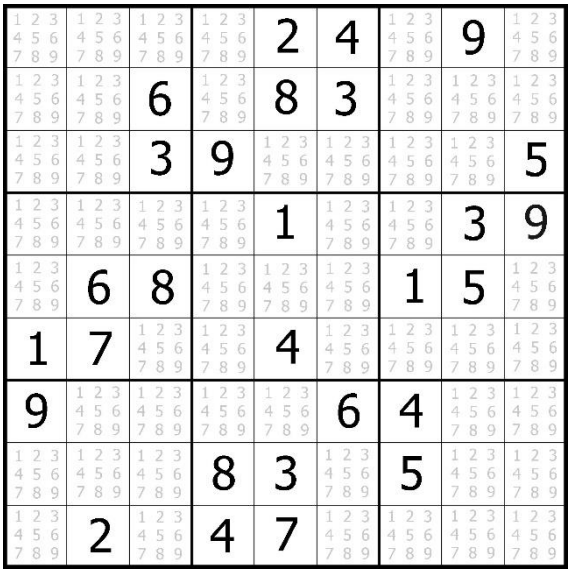


Abbildung 12: Superpositionen innerhalb eines Sudoku-Gitters

Dieses Gitter kann durch den WFCA gelöst werden. Er wählt die Zelle mit der tiefsten Entropie, beziehungsweise mit der kleinsten Ungewissheit, und bricht diese zusammen. Die Superposition wird also auf einen Zustand reduziert. Gewählt wird die Zelle mit der tiefsten Entropie, weil dort das tiefste Risiko herrscht, eine falsche Entscheidung zu treffen.

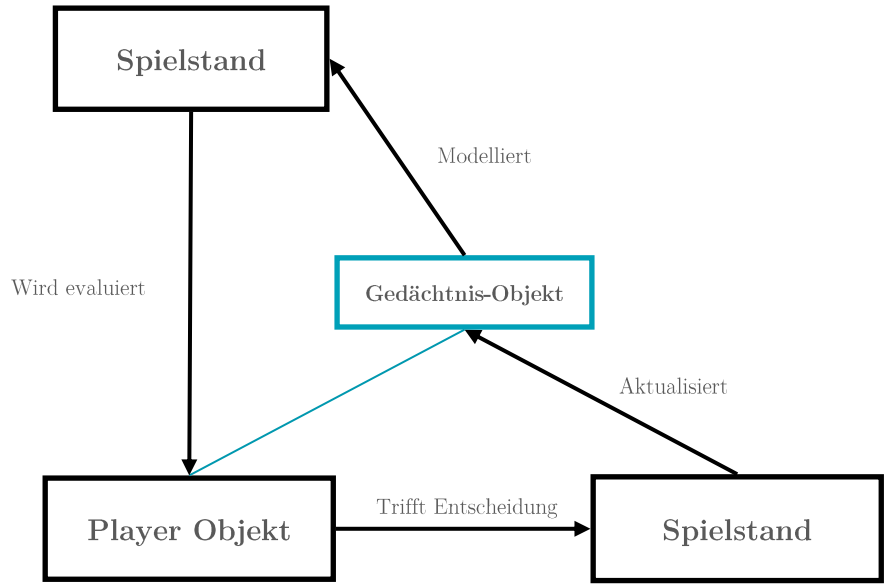
Sagt uns das Gitter, dass P1 Schaufel Acht mit einer Wahrscheinlichkeit von 0.75 besitzt, so macht es Sinn anzunehmen, dass P1 die Karte effektiv besitzt. Aus 0.9 wird mit hoher Wahrscheinlichkeit 1. Aus 0.1 wird mit hoher Wahrscheinlichkeit 0, und so weiter. Nach jedem Zusammenbruch gleicht sich das tabellarische Verzeichnis neu aus, so wie es in den obenstehenden Regeln beschrieben ist. So wird Zelle für Zelle zusammengebrochen, bis ein Gitter entsteht, dass ausschliesslich aus Nullen und Einsen besteht.

5.3.4. Der hypothetische Spielstand

Da nun die Tabelle gelöst ist, ist jeder Karte ein klarer Standort zugeschrieben, und es lässt sich ein Spielstand-Objekt modellieren. Das Einzige, was noch fehlt, ist die Ansage. Diese stützt sich ebenfalls auf die Tabelle.

Ich gehe von der Annahme aus, dass alle SpielerInnen die Faustregel für ihre Ansage verwendet haben. Das bedeutet, dass wenn all ihre Karten «bekannt» sind, ihre Ansage anhand der Faustregel erraten werden kann.

Der ganze Prozess wird durch ein Gedächtnis möglich gemacht. Jedes Player-Objekt besitzt ein Gedächtnis-Objekt, in dem alle erlangten Informationen abgespeichert werden. Nach jedem Zug wird dieses Aktualisiert.



5.3.5. Neue Evaluierungsfunktion

Ausserdem habe ich eine neue Evaluierungsfunktion eingefügt. Anstatt dass alle möglichen Kombinationen von vier Karten manuell überprüft werden, um so der durchschnittliche Stichwert jeder Karte zu bestimmen, habe ich eine Formel geschrieben, die weniger Zeitintensiv ist.

Meine Idee war es, stattdessen mit Wahrscheinlichkeiten zu rechnen. Man nimmt jede der eigenen Handkarten und vergleicht diese mit den Handkarten der anderen SpielerInnen. Dabei zählt man, wie viele Trumpfkarten und wie viele höhere Karten derselben Farbe noch im Spiel sind.

Als Beispiel nehmen wir die Karte Ecken Sechs, davon ausgehend, dass Ecken nicht Trumpf ist. Da sie die tiefste Karte ist, kann sie nur stechen, wenn sie die *erste* gespielte Karte ist[5], wenn kein Trumpf gespielt wird und keine höhere Karte derselben Farbe gespielt wird. Sagen wir, es sind noch insgesamt 9 Karten in den Händen der anderen SpielerInnen, von denen 3 weder Trumpf noch Ecken sind. Die (theoretische) Stichwahrscheinlichkeit würde so berechnet werden:

$$p_{Stich} = \frac{3}{9} \times \frac{2}{8} \times \frac{1}{7} \times \frac{1}{4} \approx 0.3\%$$

[5] Dieser Fall hat eine Wahrscheinlichkeit von einem Viertel

Diese Formel kann für Trümpfe und nicht-Trümpfe verallgemeinert werden. Für Trumpfkarten gilt das a für die Anzahl aller übrigen Karten, und das h für die höheren Trümpfe:

$$p_{Trumpf\ Stich} = \frac{a-h}{a} \times \frac{a-h-1}{a-1} \times \frac{a-h-2}{a-2}$$

Bei Karten anderer Farben muss berücksichtigt werden, dass entweder die Karte oder eine tiefere Karte derselben Farbe ausgespielt werden muss. Hier sei a Anzahl aller übrigen Karten, t für Trumpfkarten, h für Karten mit gleicher Farbe und höherem Wert, und g für Karten mit gleicher Farbe und tieferem Wert.

$$p_{Stich} = \frac{1}{4} \times \frac{a-h-t}{a} \times \frac{a-h-t-1}{a-1} \times \frac{a-h-t-2}{a-2} + \frac{3}{4} \times \frac{g}{a} \times \frac{a-h-t-1}{a-1} \times \frac{a-h-t-2}{a-2}$$

Diese Stichwahrscheinlichkeit ist aber nur theoretisch, weil sie den eingeschränkten Farbzwang nicht berücksichtigt. Sie soll vielmehr als eine grobe Schätzung dienen. Anschliessend kann die Durchschnittliche Stichwahrscheinlichkeit einer gegebenen Hand ermittelt werden, und diese mit den noch machbaren Punkten multipliziert werden. Dadurch entsteht eine neue Prognose.

5.3.6. Weitere Veränderungen

Ich habe ausserdem weitere Probleme behandelt, die mir beim Zwischenfazit im Kapitel 5.2.6 aufgefallen sind. Ich machte besonders bei der Verwendung des Kurzeitspeichers grosse Fortschritte.

Anstatt Kartenobjekte verwendete ich Indexe. Es wurden also nicht mehr mit Karten-Objekte gespielt, sondern mit 36 Zahlen, welche alle eine Karte *repräsentierten*. Das hatte eine viel kleinere Kurzspeichernutzung zur Folge.

Als zweites änderte ich das Erschaffen des Spielbaumes. Anstatt dass Spielstand-Objekte millionenfach kopiert werden, werden Spielzüge simuliert, indem innerhalb eines Spielstandes die Karten gespielt werden und diese Züge wieder rückgängig gemacht werden, bis alle Möglichkeiten ausprobiert sind. Dafür musste ich eine Warteschlage und eine «Undo»-Funktion schreiben.

Dabei bemerkte ich, dass die Knoten-Objekte, die ich ganz am Anfang eingefügt habe, überflüssig wurden. Es machte keinen Sinn mehr, die Knoten und ihre assoziierten Spielstände für später aufzubewahren, denn der Baum muss wegen den neu erlangten Informationen immer neu gebildet werden. Das gab mir die Gelegenheit, den Baum während seiner Bildung gleichzeitig zu evaluieren, ohne dass irgendwelche Nebeninformationen gespeichert werden. Die Rekursion selbst wurde zur Baumstrukur.

Um die Entscheidungen ein bisschen zu verbessern, habe ich versucht, aggressive Spielzüge zu vermeiden. Aggressive Spielzüge wären zum Beispiel, ein Nell oder ein Trumpf

Ass abzustechen, oder als aller erster Zug mit einer Trumpfkarte zu beginnen (Muff, 2022b). Solche Spielzüge wurden als aggressiv markiert, und nur mit einer reduzierten Wahrscheinlichkeit gespielt.

5.4. Umsetzung in Unity

Bild 1: Titel-Menu des Jass-Programms.

Für das Schaffen des Jass-Programms musste ich mich zuerst mit der Game-Engine Unity vertraut machen. Ich hatte anfangs grosse Schwierigkeiten, da ich mich nicht mit der Unity-Programmstruktur auskannte. Besonders die Speichernutzung stellte dabei eine grosse Herausforderung dar. Ich will mich jedoch nicht in die technischen Details dieser Implementierung verlieren, da sie nicht zum Inhalt meiner Arbeit beitragen. Ich habe den Code für die Suchalgorithmen fast unverändert in mein Unity-Projekt kopiert und diese mit den Elementen auf der Benutzerfläche verbunden. Dies ermöglichte es, einfacher mit den Suchalgorithmen zu interagieren als zuvor.

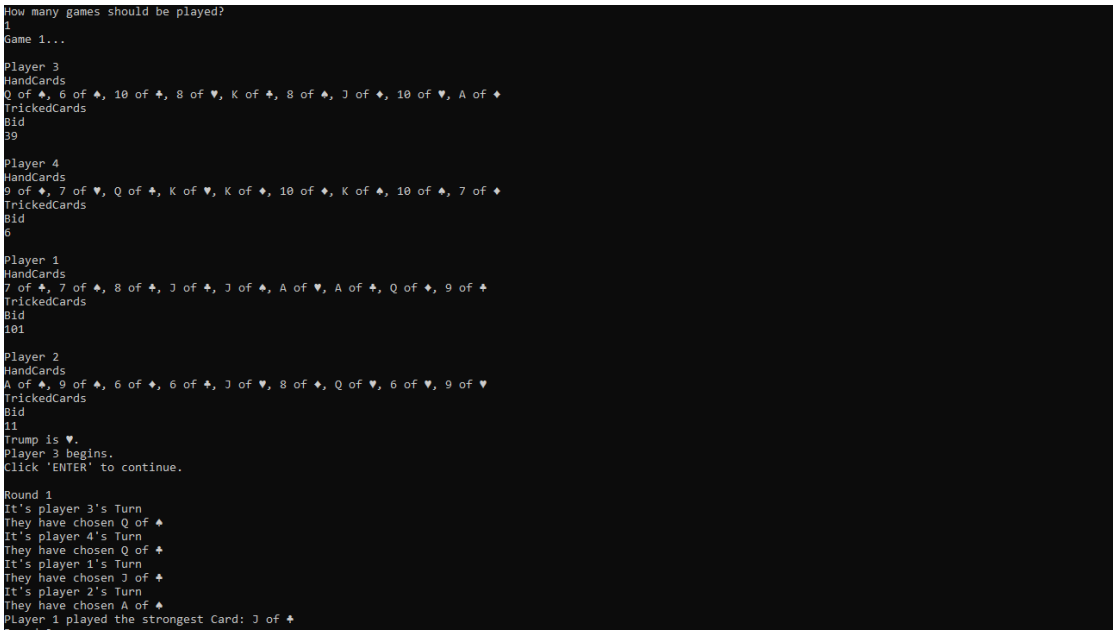


Bild 2: Bisherige Benutzeroberfläche

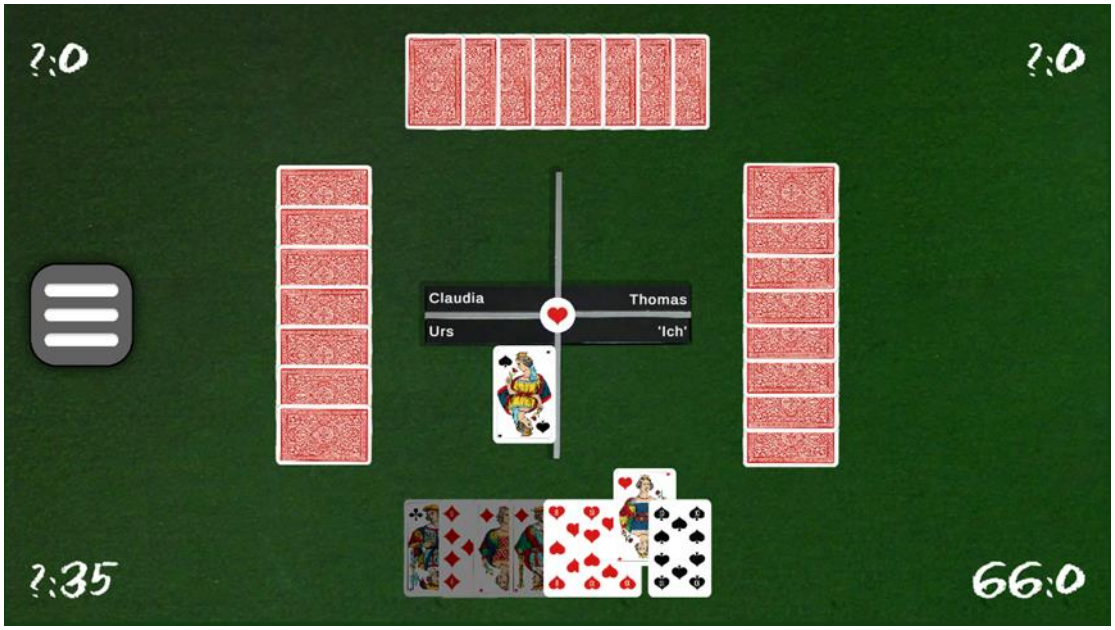


Bild 3: Benutzeroberfläche im Unity-Programm.

Das Design sollte an die Fernsehsendung «Samstig-Jass» erinnern: Als Hintergrund dient ein grüner Jassteppich, in der Mitte befindet sich ein Kreuz mit der Trumpffarbe und den Namen der SpielerInnen. Am Rand werden die Punkte der SpielerInnen angezeigt. Links am Rand befindet sich eine Menu-Leiste, mit der das Spiel pausiert werden kann.

Für die Karten-Sprites habe ein Kartenset von zuhause eingescannt und zugeschnitten. Karten können mit dem Mauszeiger ausgewählt werden. Karten, die nicht ausgewählt werden können, werden grau markiert.

Bild 4: Am Anfang des Spiels kann auf einer Jasstafel mithilfe eines Reglers die Ansage bestimmt, und mit [ENTER] bestätigt werden.

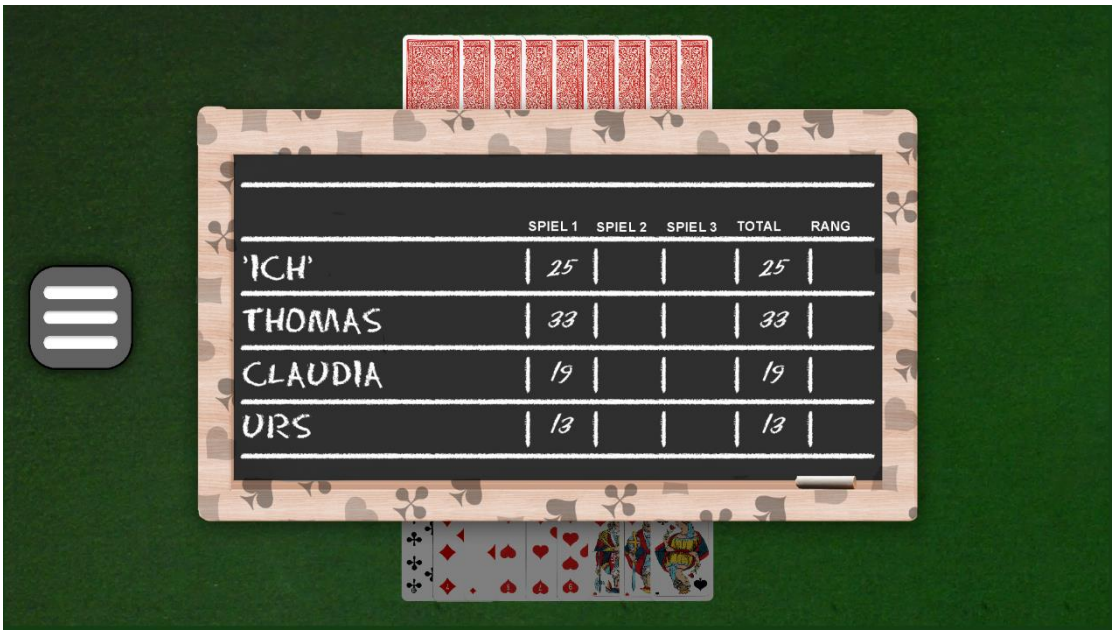


Bild 5: Am Ende jeder Runde wird der Spielstand gezeigt.

Die Jasstafeln schweben in das Spiel herein und geben Informationen. Eine Tafel gibt an, welche Farbe Trumpf ist und wer beginnt. Eine andere sagt, wer mit welcher Karte gestochen hat. Die Tafeln können nach belieben mit der [SPACE]-Taste durchsichtig gemacht werden.

Ausserdem kann innerhalb von wenigen Klicks der eigene Name, die Namen der GegnerInnen, sowie ihre Spielweise geändert werden.

Bild 6: Einstellungs-Menu. Mit einem Drop-Down-Menu kann der Suchalgorithmus der GegnerInnen geändert werden. Ein Regler bestimmt ihre Aggressivitätsschwelle.

6. Resultate und Diskussion

Immer mehr ist mir aufgefallen, dass ich bei diesem Projekt keine exakte Wissenschaft betreibe. Anders als im Schach kann ich nicht die Existenz eines besten Zuges beweisen, wie es Zermelo im Jahre 1913 gemacht hat. Ausserdem bin ich eingeschränkt durch die limitierte Rechenleistung. Ich musste deshalb grobe Vereinfachungen machen und zu eher unkonventionellen Mitteln greifen wie der WFCA, um ein funktionierendes Programm zu schaffen.

Deshalb wird es umso interessanter, die Leistung der unterschiedlichen Algorithmen empirisch zu untersuchen, um herauszufinden, ob der Computer ein ebenbürtiger Gegner sein könnte.

6.1. Rein mathematische Vergleiche

Es liegt nahe, die Algorithmen zuerst innerhalb eines Vakuums zu untersuchen, um sie «objektiv» zu bewerten. Dabei stellt sich heraus, dass aufgrund der unzähligen Einflüsse eine objektive Bewertung kaum möglich ist. Wie kann also gesagt werden, ob ein Algorithmus «gut» oder «schlecht» ist? Als unterste Messlatte kann das zufällige Spiel genommen werden.

6.1.1. Vergleich mit dem zufälligen Spiel

Um diese Frage zu beantworten, habe ich jedes Suchverfahren 1000-Mal gegen drei Computer spielen lassen, welche ihre Karten nach reinem Zufallsprinzip auswählten. Um die Messung fair zu halten, haben alle Computer ihre Ansage anhand der Faustregel gemacht, wie sie im Kapitel 5.2.1 beschrieben wird.

Algorithmus	1. Platz	2. Platz	3. Platz	4. Platz
Max^n geordnet	282	256	246	216
Max^n	294	254	240	212
Paranoia geordnet	300	249	252	199
Paranoia	278	250	255	217

Tabelle 12: Die Platzverteilung der unterschiedlichen Suchverfahren nach 1000 zufälligen Spielen.

Um eine Statistische Auswertung zu machen, braucht es zuerst eine Nullhypothese, welche geprüft werden soll. Die Nullhypothese lautet in diesem Fall: Zwischen den Suchalgorithmen und ihren Leistungen besteht kein Zusammenhang, folglich werden Suchalgorithmen im Durschnitt gleich abschneiden wie das zufällige Spiel.

Das bedeutet, dass die Gewinnwahrscheinlichkeit 25% beträgt, und der Erwartungswert der gewonnenen Spiele 250 ist. Was aber auffällt: Alle vier Suchverfahren gewinnen mehr als 250 Mal. Hier lohnt sich ein Blick auf die Normalverteilung[6] der Gewinne.

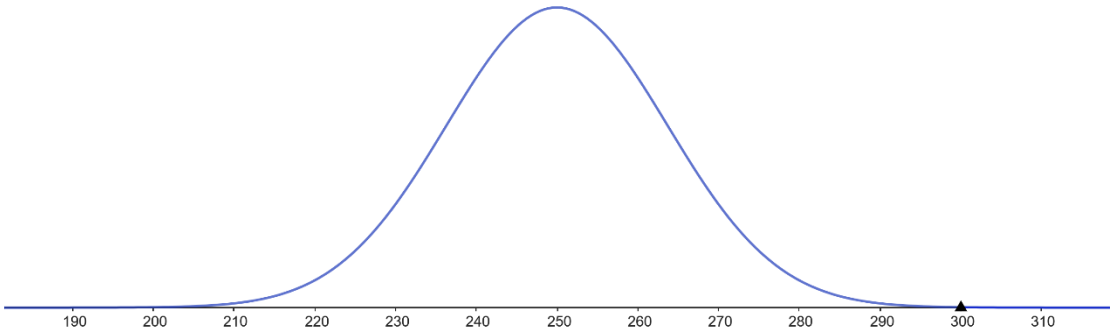


Abbildung 13: Normale Verteilkurve der Anzahl Gewinne bei 1000 Spielen. Auch hier stellen 250 Gewinne die höchste Wahrscheinlichkeit dar.

Anhand des Graphen ist ersichtlich, dass es höchst unwahrscheinlich ist (p-Wert = 0.01), 300-mal oder mehr zu gewinnen, wenn man davon ausgeht, dass die Suchalgorithmen gleich gut sind wie die zufällige Wahl. Zwar kann der p-Wert keine direkte Aussage über die Wahrheit der Hypothese und die Qualität der Algorithmen sagen, da aber der p-Wert so schwindend klein ist, können wir mit gutem Gewissen die Nullhypothese verwerfen und uns stattdessen darauf einigen, dass die Algorithmen signifikant besser abschneiden als das zufällige Spiel.

Algorithmus	1. Platz	p-Wert
Max^n geordnet	282	0.97%
Max^n	294	0.03%
Paranoia geordnet	300	0.01%
Paranoia	278	2.14%

Tabelle 13: Anzahl Gewinne und korrespondierende p-Werte der unterschiedlichen Suchalgorithmen.

Dass der geordnete Paranoia-Algorithmus so gut abschneidet, überrascht kaum, wenn man bedenkt, dass er sich auf das Worst-Case-Szenario einstellt und seinen eigenen Rang priorisiert. Das zufällige Spiel kann folglich nur besser sein als das Worst-Case-Szenario.

[6] Hier wäre die binomische Verteilung angemessener gewesen, da gewinnen und verlieren (gewinnen meint erstplatziert sein) binäre zustände sind, und somit die Verteilfunktion eigentlich diskret sein müsste. Aufgrund der hohen Anzahl der Proben kann jedoch problemlos die Normalverteilung (oder Gauss-Verteilung) als Annäherung der Gewinnverteilung genommen werden.

Dass die beiden Max^n-Algorithmen im Vergleich mit dem normalen Paranoia-Algorithmus so gut abschneiden, ist jedoch sehr verwunderlich, da sie mit ihren optimistischen Annahmen falsch liegen.

6.1.2. Vergleich untereinander

Der Vergleich zwischen den Suchalgorithmen gestaltete sich um einiges schwieriger als der Vergleich mit dem zufälligen Spiel. Erstens waren die Simulationen sehr ressourcenintensiv, da nicht nur ein Computer einen Suchalgorithmus verwendet, sondern alle vier. Aus diesem Grund reduzierte ich die Probengrösse von Tausend auf Hundert.

Ein anderes Problem war die überwältigende Anzahl Kombinationen. Da die Spielreihenfolge möglicherweise ein Einflussfaktor sein könnte, habe ich jede Permutation einzeln geprüft. Bei 4^4 Kombinationen à 100 Spielen sind das insgesamt 25'600 simulierte Spiele.

Als Beispiel könnte man die Konstellation Paranoia (P), geordneter Paranoia (Pg), Max (M) und geordneter Max (Mg) betrachten. Für diese Kombination gibt es insgesamt 24 Permutationen. Man könnte annehmen, dass die Permutationen ähnliche Resultate zurückgeben.

	1st	2nd	3rd	4th
Mg	23	24	28	25
M	31	20	24	25
Pg	20	25	32	23
P	31	30	19	20

	1st	2nd	3rd	4th
P	31	30	16	23
Mg	23	25	34	18
M	24	25	22	29
Pg	29	19	22	30

	1st	2nd	3rd	4th
P	21	22	23	34
Pg	23	25	30	22
M	30	22	28	20
Mg	34	25	19	22

	1st	2nd	3rd	4th
Pg	40	18	27	15
M	20	21	30	29
Mg	27	30	20	23
P	20	27	21	32

Tabelle 14: Platzverteilungen von vier unterschiedlichen Spielreihenfolge. Tabellen nebeneinander haben eine verschobene Reihenfolge, Tabellen untereinander haben eine umgekehrte Reihenfolge derselben Permutation. Farbig markierte Zellen weichen stark vom Erwartungswert 25 ab.

Es wird ersichtlich, dass die unterschiedlichen Reihenfolgen komplett andere Resultate erzeugen. Die Beispiele, die ich hier aufzeige, sind ausserdem nur ein kleiner Bruchteil aller simulierten Spiele. Es wäre praktisch unmöglich, alle 256 Permutationen auf diese Art zu vergleichen, um herauszufinden, welches der beste Algorithmus ist.

Es ist unklar, ob die Reihenfolge effektiv einen Einfluss auf die Gewinnchancen hat, oder ob es nicht eher die zufällige Verteilung der Karten ist, welche die Resultate so unterschiedlich macht. Diese Zweifel eröffnen Türen für weitere Fragen, zum Beispiel, ob die eigene Hand die Leistung der Suchalgorithmen beeinflusst. Zwar sollten die eigenen

Karten in Theorie nicht die eigenen Gewinnchancen beeinflussen, weil mit der richtigen Ansage jedes Spiel gewinnen werden könnte, aber ob diese Hypothese auch wirklich stimmt, bleibt offen.

Auch andere Daten wie durchschnittliche Differenz geben uns kaum Auskunft auf die Leistung, da die Werte alle sehr ähnlich sind, und die Varianzen sehr gross. Das grösste Problem ist, dass es keinen wahren Anhaltspunkt gibt, auf dem man sich stützen könnte.

Lösen könnte man einige dieser Unsicherheiten, indem man bessere Test-Bedingungen geschafft hätte. So hätte ich stattdessen immer nur eine Kartenverteilung mit unterschiedlichen Anordnungen von Algorithmen prüfen können. Hätte ich ausserdem mehr Simulationen durchführt, was jedoch sehr lange gedauert hätte, wären die Ergebnisse statistisch signifikanter geworden.

Allgemein wird klar, dass der direkte Vergleich zwischen den Algorithmen sehr unproduktiv ist. Es gibt zu viele unvorhersehbare Parameter, welche die Leistung eines Algorithmus beeinflussen können. Ausserdem besteht die grosse Schwierigkeit, dass die Algorithmen von der falschen Annahme ausgehen, und so gewissermassen zum Scheitern verdammt sind. Spielen alle paranoid, so kann die Rechnung grundsätzlich nicht aufgehen.

Das Spiel könnte nur richtig funktionieren, wenn alle nach dem Max^n-Prinzip spielen. Dort gilt die Annahme, dass alle SpielerInnen nur ihren eigenen Punktestand maximieren möchten, und die anderen Differenzen nicht beachten.

	1st	2nd	3rd	4th	Durchschnitt	Median	Varianz	Standartabweichung
1	26	24	24	26	18.13	16.5	209.39	14.47
2	27	21	29	23	18.21	14	202.13	14.22
3	32	31	19	18	16.59	13	198.77	14.10
4	21	23	26	30	19.83	17	251.48	15.86

Tabelle 15: Platzierung und Differenzen bei 100 Spielen mit dem Max^n.

	1st	2nd	3rd	4th	Durchschnitt	Median	Varianz	Standartabweichung
1	23	22	25	30	21.83	16.5	358.97	18.94
2	26	31	22	21	19.82	15.5	272.69	16.51
3	33	17	27	23	18.56	14.5	193.36	13.90
4	25	28	23	24	19.04	15	226.66	15.05

Tabelle 16: Platzierung und Differenzen bei 100 Spielen mit dem geordneten Max^n.

Auffallend ist, wie ungleichmässig die Differenzen verteilt sind, der dritte Computer scheint dabei im Vorteil zu sein, der erste schneidet hingegen viel schlechter ab. Ausserdem schneidet der Max^n bezüglich der Differenzen besser ab als der geordnete. Das kann dadurch erklärt werden, dass der ungeordnete Algorithmus die absolute Differenz

priorisiert und diese versucht zu vermindern, wohingegen der geordnete Algorithmus primär auf die Platzierung in der Rangliste achtet, was zu grösseren Differenzen führt.

Allgemein lässt sich sagen, dass die Differenzen im Schnitt etwa 50% grösser sind als beim offenen Differenzler (siehe Tabelle 9). Das scheint zwar logisch, wenn man berücksichtigt, dass am Anfang des Spiels 75% der Informationen fehlen, aber es offenbart auch die Grenzen und nichtexistierende Intuition dieser Algorithmen.

6.2. Vergleich mit dem Menschen

Die ersten Tests zeigen, dass die Algorithmen zwar besser sind als eine zufällige Auswahl der Karten, aber im Vakuum nicht besonders gut spielen. Ausserdem entsteht die Schwierigkeit, die Algorithmen ohne wahren Bezugspunkt zu bewerten.

Um klarere Aussagen zu machen, müssen die Algorithmen mit Menschen verglichen werden. Dieser Vergleich erfolgt auf zwei Arten: Erstens mit der statistischen Auswertung und dem Vergleich von menschlichen Spielen, zweitens mit der direkten Konfrontation zwischen Menschen und Computer.

6.2.1. Tourniere

Bevor ich Mensch und Computer gegeneinander spielen liess, wollte ich die Resultate der Tabelle 16 relativieren. Aus diesem Grund habe ich Rund 250 Spiele vom «Samstig» und «Donnstig»-Jass statistisch ausgewertet und miteinander verglichen.

Durchschnitt	Median	Varianz	Standartabweichung
9.18	6.5	81.26	9.01

Tabelle 17: Differenzen bei menschlichen SpielerInnen

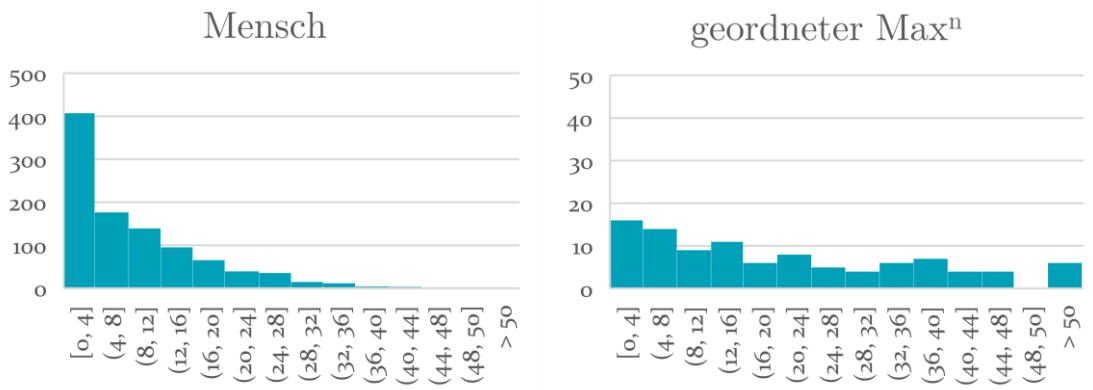


Abbildung 14: Verteilung der Differenzen bei menschlichen SpielerInnen und beim Max^n Algorithmus

Die menschlichen SpielerInnen haben im Durchschnitt etwa halb so grosse Differenzen wie der Computer, ausserdem sind die Differenzen weniger stark verstreut. Die

Differenzen sind bei den Menschen logarithmisch verteilt, Differenzen von Null Punkten sind sehr häufig, wohingegen grosse Differenzen nur sehr selten geschehen.

Der geordnete Max^n sieht dabei nicht besonders gut aus. Visuell ist ersichtlich, dass die Differenzen viel verstreuter sind. Dasselbe gilt auch für den normalen Max^n- und den geordneten Paranoia-Algorithmus, wobei der normale Paranoia-Algorithmus wegen seiner Verteilkurve auffällt:

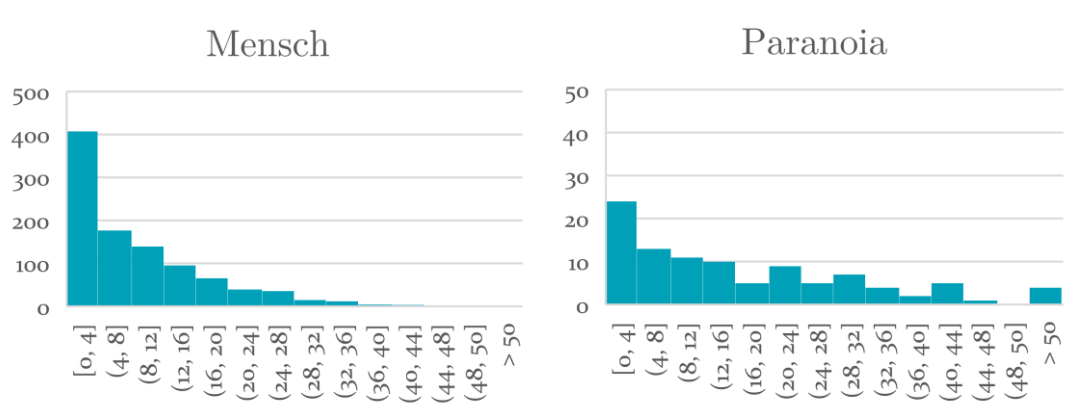


Abbildung 15: Verteilung der Differenzen beim Paranoia-Algorithmus

Von allen Verteilkurven sieht die des Paranoia-Algorithmus ähnlichsten aus. Doch auch hier ist die Streuung der Ergebnisse um einiges grösser, und so auch die durchschnittliche Differenz. So kann gesagt werden, dass Menschen auf dem Niveau des «Samstig»- und «Donnstig»-Jass untereinander besser spielen als der Computer.

6.2.2. Ein Mensch gegen drei Computer

Um die Leistung der Algorithmen im direkten Vergleich mit einem menschlichen Spieler zu messen, ließ ich mein Jassprogramm gegen meinen Vater antreten, der sich selbst weder als Anfänger noch als Profi einstuft.

	Durchschnitt	Median	Varianz	Standartabweichung
Durchschnitts-Jasser	8.12	4	89.24	9.45
Max^N	23.75	22.5	312.39	17.67
Paranoia	19.12	18	154.24	12.42
Geordneter Paranoia	19.35	11	360.49	18.99

Tabelle 18: Differenzen, Varianzen und Standartabweichungen bei der direkten Konfrontation von Mensch und Computer nach 17 Spielen

Die Ergebnisse zeigen, dass die Suchalgorithmen deutlich schlechter abschneiden als der menschliche Spieler. Die durchschnittlichen Differenzen sind etwa doppelt so gross und auch die Varianzen fallen wesentlich höher aus.

Bereits in vorherigen Vergleichen zeigte sich, dass die Leistung der Suchalgorithmen eher schwach ist. Gegen einen menschlichen Gegner erhöht sich der Durchschnitt der Differenzen weiter.

Meiner Einschätzung nach könnte die Spielweise meines Vaters einen Einfluss auf diese Ergebnisse haben. Seine Neigung zu Null-Ansagen, selbst wenn er Asse auf der Hand hatte, führte möglicherweise dazu, dass zu wenige Punkte angesagt wurden. Dies könnte zu höheren Differenzen im Gesamtspiel geführt haben.

7. Fazit und Ausblick

Letztendlich erwiesen sich die untersuchten Suchalgorithmen als überlegen gegenüber rein zufälligen Kartenentscheidungen. Doch trotz dieser Überlegenheit fallen ihre Leistungen unzureichend aus. Mehrere Faktoren tragen dazu bei: Die mathematische Unlösbarkeit des Jassspiels macht es schlicht unmöglich, den optimalen Spielzug zu berechnen. Zudem gestaltet sich das Treffen langfristig effizienter Entscheidungen bei begrenzter Suchtiefe als äußerst schwierig. Einflüsse wie die Verteilung der Karten oder die Reihenfolge des Spiels könnten ebenfalls die Algorithmen beeinflussen. Es besteht sogar die Möglichkeit, dass der Maxn- und Paranoia-Algorithmus generell ungeeignet für das Jassspiel sind.

Das alles bestätigt, dass erfolgreiches Jassen vor allem auf Erfahrung und Intuition beruht. Trotz allem bin ich zufrieden mit meiner Arbeit, da ich mit reiner Mathematik schon sehr vieles erreichen konnte. Die Tatsache, dass die Suchalgorithmen besser abschneiden als zufällige Entscheidungen, unterstreicht die gewisse Vorhersehbarkeit des Spiels.

Gerne hätte ich mich weiter in das Thema vertieft. Es wäre spannend gewesen, die Algorithmen gegen drei menschliche Spieler oder in Dreispieler-Szenarien zu testen. Bedauerlicherweise fehlte mir dafür die Zeit. Ebenso wäre eine detailliertere Untersuchung unterschiedlicher Suchtiefen oder Aggressivitätsgrade der Algorithmen interessant gewesen. Eine bessere Teststruktur für die Algorithmen hätte außerdem die Vergleiche zwischen ihnen vereinfacht.

Klar ist, dass im Differenzler noch sehr viel Potenzial für weitere Forschungen steckt. Besonders bei der heuristischen Evaluierung von Spielständen gibt es noch viel zu klären, denn die Frage, wie man anhand eines Blattes eine Punktzahl vorhersagen kann, bleibt noch offen. Ich glaube, diese Frage bildet das Herzstück meiner Arbeit. Ein möglicher Lösungsansatz könnte die Entwicklung eines maschinellen Lernverfahrens für die Ansage und Evaluierung von Blattknoten sein. Auch eine tiefere Erforschung des Monte-Carlo-Algorithmus wäre vielversprechend.

Während meiner Arbeit konnte ich mich gleichzeitig in so viele Bereiche vertiefen, die mich interessierten: Ich erlernte eine neue Programmiersprache, stieß auf Herausforderungen wie die effiziente Nutzung von Speicherplatz, eignete mir Wissen über Spieltheorie und Suchalgorithmen an und setzte dieses Wissen praktisch ein, ich machte mich mit Spielentwicklung vertraut und konnte ein Programm gestalten, all dies wärenddem ich meinem Interesse für Kartenspiele nachgehen konnte.

8. Literaturverzeichnis

Bärtschi, A. (2011). *Spieltheorie.* Mentorierte Arbeit in fachwisssenschaftlicher Vertiefung, ETH, Mathematik, Zürich.

Bewersdorff, J. (1998). *Glück, Logik und Bluff: Mathematik in Spiel - Methoden, Ergebnisse und Grenzen* (1998 Ausg.). Wiesbaden: Vieweg & Teubner.

Copeland, B. J. (16. Dezember 2008). *Stanford Encyclopedia of Philosophy.* Von The Modern History of Computing: https://plato.stanford.edu/entries/computing-history/ abgerufen

Donald, M. (31. Juli 2020). Superpositions, Sudoku, the Wave Function Collapse algorithm. Abgerufen am 14. Dezember 2023 von https://www.youtube.com/watch?v=2SuvO4Gi7uY

IBM. (kein Datum). Abgerufen am 2. Januar 2024 von Deep Blue: https://www.ibm.com/history/deep-blue

Luckhardt, C. A., & Irani, K. B. (1986). *An algorithmic solution of n-person games.* University of Michigan.

Muff, A. (19. 9 2022a). *Jassregeln: Wie geht der Differenzler?* Von Schweizer Jassverzeichnis: https://jassverzeichnis.ch/differenzler/ abgerufen

Muff, A. (Mai 2022b). *Jass-Tipps: Tricks & Tipps für den Differenzler-Jass.* Von jassverzeichnis.ch: https://jassverzeichnis.ch/jass-tipps-differenzler/ abgerufen

Nijssen, J. A. (2013). *Monte-Carlo Tree Search for Multi-Player Games.*

Ortmanns, W., & Albert, A. (2008). *Entscheidungs- und Spieltheorie: Eine anwendungsbezogene Einführung* (1 ed.). Sternenfels: Wissenschaft & Praxis.

Romer, K. (8. Januar 2022). How A.I. Conquered Poker. *New York Times.* Abgerufen am 29. Oktober 2023 von https://www.nytimes.com/2022/01/18/magazine/ai-technology-poker.html

Shmueli, T., & Zuckerman, I. (2015). *Avoiding Game-tree Pathology in Multi-Player Games.*

Staffin, P. D. (1996). *Game Theory and Strategy.* Washington, DC, USA: American Mathematical Society.

Winands, M. H. (2011). *Best Reply Search for Multiplayer Games.*

Zermelo, E. (1913). *Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels.* Cambridge. Retrieved from http://webdocs.cs.ualberta.ca/~hayward/396/asn/zermelo.pdf

9. Anhang

Der C#-Quellcode der unterschiedlichen Algorithmen kann unter folgendem Link aufgerufen, heruntergeladen und bearbeitet werden: https://github.com/tlb0/Jass-Programm