Henriette Schulz
Mathias-Hess-Straße 80
69190 Walldorf
schulz.henriette.0106@gmail.com
Klasse: J1/3
Abgabedatum: 08. März 2023

Python im Alltag

Vorstellung eines selbstgewählten, komplexen Algorithmus am Beispiel des Spiels „Vier gewinnt“

Hausarbeit
vorgelegt zur Gleichwertigen Feststellung von Schülerleistungen (GFS)
an der Johann Philipp Bronner Schule Wiesloch

Kurs: Informatik

Betreuer: Benjamin Speckert

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Zielstellung

Das Ziel dieser Arbeit besteht darin, ein Alltagsproblem mit Hilfe eines komplexen Algorithmus zu beschreiben und mit Python zu implementieren. Dabei soll – entsprechend der im Informatikkurs vermittelten Grundlagen – prozedurale und keine objektorientierte Programmierung zu Einsatz kommen.

1.2 Zu lösendes Alltagsproblem

Das bekannte Gesellschaftsspiel „Vier gewinnt“ soll mittels eines Computerprogramms nachgebildet werden. Bei diesem Spiel lassen abwechselnd zwei Spieler Steine von oben auf ein sieben mal sechs Felder großes Spielfeld fallen. Das Ziel des Spiels besteht darin, als erster Spieler vier zusammenhängende Felder – in einer Zeile, Spalte oder Diagonale – zu erreichen[1].

In diesem Projekt soll das Spiel in grafischer Art und Weise dargestellt werden und es ermöglichen, dass sowohl zwei menschliche Spieler gegeneinander spielen als auch ein menschlicher Spieler gegen den Computer spielt.

Für die Lösung dieses komplexen Problems sind an verschiedenen Stellen Algorithmen erforderlich, die einen unterschiedlichen Komplexitätsgrad aufweisen.

2 Grundlegender Ablauf und schrittweise Verfeinerung

Im folgenden Abschnitt soll gezeigt werden, wie sich das Problem des Spiels in einem grundlegenden Algorithmus umsetzen lässt, der wiederum schrittweise verfeinert wird, um weitere Teilprobleme lösen zu können.

2.1 Das Spielfeld

Um das Spiel mit einem Computer nachbilden zu können, soll zunächst das Spielfeld in einer Form beschrieben werden, die dafür geeignet ist. Das Spielfeld, das aus sieben mal sechs Feldern besteht, stellen wir uns als Matrix vor, die sieben Spalten und sechs Zeilen besitzt. Diese Zeilen und Spalten sind jeweils von null beginnend nummeriert. Wenn ein Feld mit einem Spielstein des ersten Spielers belegt ist, soll dort eins stehen, für den zweiten Spieler zwei und sonst null.

---

[1] Zu Spielregeln siehe: (Hasbro SA, 2020). Zur Geschichte des Spiels siehe: (wikipedia - vier gewinnt, 2022)

Abbildung 2.1: Darstellung des Spielfeldes als Matrix

	0	1	2	3	4	5	6
5	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0
2	0	0	0	0	1	2	0
1	0	0	2	2	1	2	0
0	1	2	2	1	1	1	2

2.2 Grundidee

Das Spiel wird von zwei Spielern abwechselnd gespielt, indem sie Züge machen. Dies wird so lange fortgesetzt, solange noch Züge möglich sind und keine Gewinnposition eines Spielers erreicht wurde. Der entsprechende Algorithmus ist im Anhang 6.2.1.1 dargestellt, das zugehörige Struktogramm in Abbildung 6.1. Das Ausführen des Zuges kann dabei sowohl durch einen menschlichen Spieler als auch durch den Computer erfolgen.

Um diese Schritte durch den Computer ausführen zu lassen, müssen weitere Teilprobleme gelöst werden: Es muss geprüft werden, ob eine Gewinnposition vorliegt. Weiterhin muss geprüft werden, ob noch Züge ausführbar sind. Schließlich muss der Computer einen Vorschlag für einen Zug ermitteln.

2.3 Prüfung auf eine Gewinnposition

Beim Spiel muss nach jedem Zug geprüft werden, ob eine Gewinnposition vorliegt. Gewinnpositionen können dabei in einer Zeile, einer Spalte, einer Diagonale von rechts oben nach links unten und in einer Diagonale von links oben nach rechts unten auftreten.

Um für eine Zeile ermitteln zu können, ob eine Gewinnposition vorliegt, soll nun für jedes Feld innerhalb der Zeile geprüft werden, ob es vom Spieler belegt ist. Wenn das Feld vom Spieler belegt ist, soll mitgezählt werden – wenn nicht, wird der Zähler wieder auf null gesetzt. Falls der Zähler vier erreicht, wurde eine Gewinnposition in der Zeile festgestellt. Der sich ergebende Algorithmus wird im Anhang 6.3.2.2 verbal und als Struktogramm (siehe dort Abbildung 6.4) beschrieben.

Für die Ermittlung einer Gewinnposition in einer Spalte wird der Algorithmus entsprechend angepasst.

Für die Ermittlung der Gewinnposition in einer Diagonalen ist zunächst noch das Problem zu lösen, dass alle zu untersuchenden Felder einer Diagonalen festgestellt werden müssen. Hierzu wird – ausgehend vom aktuellen Zug und seiner Position – als erstes die Position am Rand ermittelt und von dort aus entlang der Diagonalen iteriert, bis wieder der Rand erreicht wurde.

Am Ende ist eine Gewinnposition dann erreicht, wenn in einer Zeile, Spalte oder in einer der Diagonalen eine Gewinnposition gefunden wurde.

2.4 Ermittlung der ausführbaren Züge

Eine weitere grundlegende Funktion besteht darin, dass festgestellt werden soll, ob überhaupt noch Züge ausgeführt werden können. Die grundlegende Lösungsidee für dieses Teilproblem besteht darin, in jeder Spalte von oben zu schauen, ob es noch freie Felder gibt. Wenn dies der Fall ist, kann noch weitergespielt werden und das Spiel ist noch nicht zu Ende.

Der gewählte Algorithmus kann eine Liste der freien Felder ermitteln (Anmerkung: Natürlich könnte man den Algorithmus bereits beim ersten Auftreten einer freien Zelle beenden. Allerdings werden wir später noch Funktionen benötigen, die uns alle möglichen Züge liefern, so dass das hier gleich berücksichtigt wird.). Der entsprechende Algorithmus wird im Anhang 6.3.2.1 verbal und dort in Abbildung 6.3 als Struktogramm beschrieben.

3 „Künstliche Intelligenz“ für einen Computerspieler

Mit den bisher beschriebenen grundlegenden Algorithmen ist es möglich, das Spiel zweier menschlicher Spieler auf einem Computer zu beschreiben. Allerdings ist die Erwartung an ein Computerspiel, dass der Computer als Gegner fungieren kann und dabei eine gewisse „Intelligenz“ mitbringt. Im vorliegenden Fall soll der Computer in der Lage sein, auf der Grundlage der jeweiligen Spielstellung sinnvolle Züge durchzuführen.

3.1 Spieltheoretische Grundlagen

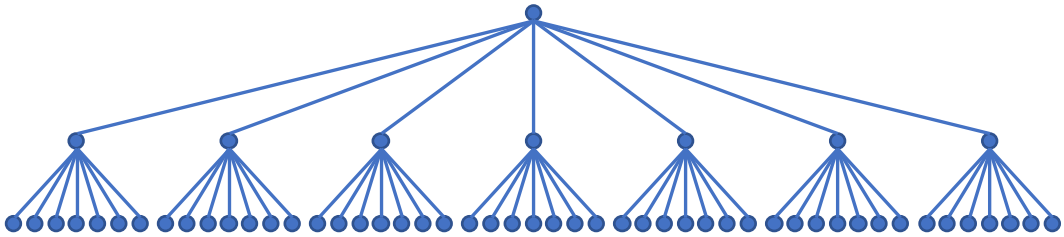
Spiele sind von Mathematikern bereits seit geraumer Zeit untersucht worden. Die moderne mathematische Spieltheorie wurde von John von Neumann begründet.[2] Im Sinne der Spieltheorie handelt es sich bei „Vier gewinnt“ um ein sogenanntes *Zwei-Personen-Nullsummenspiel*. Das ist ein Spiel, bei dem die Summe der Auszahlungen an beide

[2] Siehe dazu: (Neumann, 1928). Siehe dazu außerdem eine Abhandlung darüber in (Kjeldsen, 2001).

Spieler immer Null ergibt. Anders ausgedrückt: In der Endstellung ist der Nutzwert für den einen Spieler genau so groß wie der negative Nutzwert des anderen Spielers. Damit kann es zwischen beiden Spielern keine gewinnbringende Kooperation geben.

Außerdem handelt es sich um ein Spiel mit *voller Information*.[3] Das bedeutet, dass allen Spielern jederzeit alles über den Spielstand bekannt ist, so dass alle Entwicklungsmöglichkeiten des Spiels lückenlos vorhergesagt werden können.[4]

Abbildung 3.1: Spielbaum nach einer Runde



Das Spiel lässt sich über einen Spielbaum analysieren. Ein vollständiger Spielbaum wird sehr schnell sehr groß. Bereits nach nur einer Runde hat er $7 * 7 = 49$ Äste. Damit ergeben sich maximal $7^{21} \approx 5{,}6 * 10^{17}$ Zustände (maximal 21 Runden mit je sieben verschiedenen Möglichkeiten; real weniger, da am Ende nicht immer sieben Zugmöglichkeiten vorhanden sind). Nach einer vorliegenden Untersuchung beträgt die Gesamtzahl der Spielstellungen bei „Vier Gewinnt“ rund 4,5 Billionen.[5]

Im Jahr 1988 haben Victor Allis[6] und James D. Allan[7] unabhängig voneinander gezeigt, dass „Vier Gewinnt“ vollständig gelöst ist und es für den beginnenden Spieler immer möglich ist zu gewinnen.[8]

---

[3] Fußball ist im Gegensatz dazu kein Spiel mit voller Information. Beispielsweise könnte der Torwart plötzlich durch die Sonne geblendet werden oder ein plötzlicher Windstoß die Richtung des Balles verändern.
[4] Aus: (Krusenotto, 2016, S. 125 ff.)
[5] Siehe: (Edelkamp & Kissmann, 2008)
[6] Siehe: (Allis, 1988)
[7] Siehe: (Allen, 2010)
[8] Siehe: (Ruile, Weiß, Ditsche, & Schmidt, 2009)

3.2 Der MinMax-Algorithmus

3.2.1 Motivation

Nullsummenspiele mit voller Information lassen sich durch den *MinMax-Algorithmus* (oft auch Minimax-Algorithmus genannt) analysieren.[9] Diese Spielbäume lassen sich mit einem geeigneten Algorithmus durchsuchen, wobei jeweils der Zug ausgewählt werden soll, der für den betrachteten Spieler den größten Nutzen bringt. John von Neumann gilt als derjenige, der dieses Prinzip erstmals wissenschaftlich untersucht hat.[10]

Eine vollständige Untersuchung dieser Suchbäume ist für gängige Personalcomputer allerdings zu aufwändig. Die entstehenden Laufzeiten wären für ein interaktives Spiel viel zu lang. Eine vorherige Berechnung einschließlich ihrer Abspeicherung der Spielzustände würde ebenfalls die Grenzen der heimischen Computer sprengen. Anstatt alle Stellungen zu untersuchen, stoppt die Berechnung bei einer bestimmten Tiefe des Spielbaumes und kann darum – basierend auf einer geeigneten Bewertung des jeweiligen Spielstandes – nur einen kleineren Teil des Baumes durchsuchen.

3.2.2 Exkurs: Rekursion und Tiefensuche

Um Bäume durchsuchen zu können, sind verschiedene Suchverfahren bekannt. Bei der *Tiefensuche* wird zunächst ein Pfad vollständig in die Tiefe beschritten, bevor die abzweigenden Pfade beschritten werden.[11] Eine Alternative dazu stellt beispielsweise die *Breitensuche* dar, bei der zunächst alle Knoten untersucht werden, die vom Ausgangsknoten direkt erreichbar sind, ehe die Folgeknoten beschritten werden.[12]

Diese Verfahren können sehr gut durch *Rekursion* beschrieben werden. Bei der rekursiven Programmierung ruft sich eine Funktion in einem Computerprogramm selbst wieder auf. Auch ein gegenseitiger Aufruf von zwei Funktionen stellt eine Rekursion dar. Wichtig ist dabei eine Abbruchbedingung, da sich sonst die Funktion unendlich oft aufrufen würde (und damit irgendwann die Speichergrenzen des Computers sprengen würde).[13]

---

[9] Ausführliche Betrachtungen dazu finden sich u.a. in (Adorf, 2009). Da es in dieser Arbeit in erster Linie um die Umsetzung des Algorithmus in Python und nicht um die Herleitung und nähere Untersuchung desselben gehen soll, wird sich an dieser Stelle auf die Darstellung des Algorithmus beschränkt und auf weiterführende Literatur verwiesen.

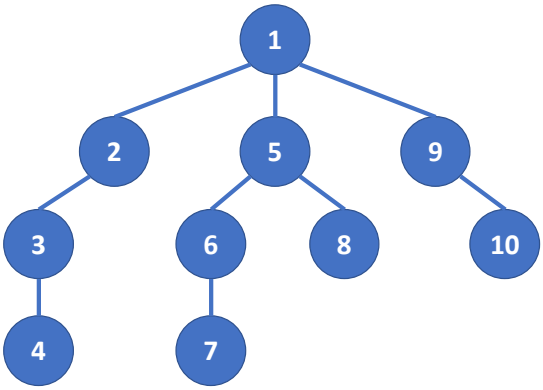
[10] Nach (Kjeldsen, 2001) hat John von Neumann dies als erstes in (Neumann, 1928) dokumentiert.

[11] Siehe dazu: (wikipedia - Tiefensuche, 2022)

[12] Siehe dazu: (wikipedia - Breitensuche, 2022)

[13] Siehe dazu: (wikipedia - rekursive Programmierung, 2022)

Abbildung 3.2: Suchreihenfolge bei der Tiefensuche in Bäumen[14]



3.2.3 Allgemeiner Aufbau des MinMax-Algorithmus

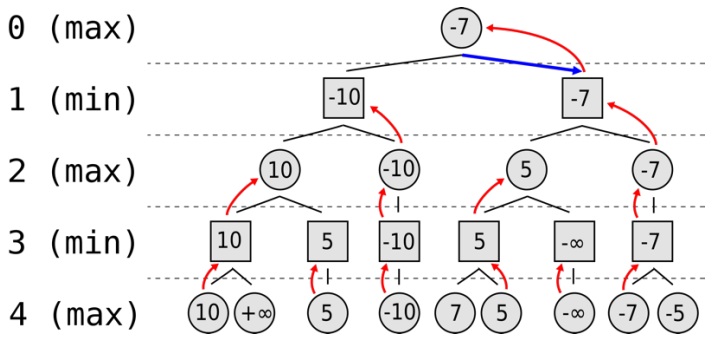
Das unmittelbare Ziel des MinMax-Algorithmus besteht darin, die MinMax-Bewertung für die aktuelle Stellung zu berechnen. Für den Spieler Max bedeutet ein positiver MinMax-Wert einen Gewinn. Für alle Endstellungen ist der MinMax-Wert gleich dem Nutzwert für Max. Der Nutzwert steht dafür für die Bewertung der Stellung aus Sicht des jeweiligen Spielers. Während der Algorithmus für alle geeigneten Spiele gleich ist, unterscheidet sich die Nutzwertfunktion von Spiel zu Spiel (eine Stellung im Schach ist offensichtlich nicht direkt mit einer Stellung bei „Vier Gewinnt“ vergleichbar).

Der Algorithmus kann damit wie folgt definiert werden:

$$minmax(r) = \begin{cases} nutzwert(r), falls\ r\ Endzustand \\ max\{minmax(s) \mid s \in \Gamma(r)\}, falls\ r\ Max-Knoten \\ min\{minmax(s) \mid s \in \Gamma(r)\}, falls\ r\ Min-Knoten \end{cases}$$

Dabei soll die Funktion Γ alle Nachfolgestellungen im Spielbaum liefern. Eine Implementierung des Algorithmus liefert nicht nur den MinMax-Wert zurück, sondern auch den Zug, der zu diesem Wert geführt hat.[15]

Abbildung 3.3: Anwendung des MinMax-Algorithmus in einem Spielbaum[16]

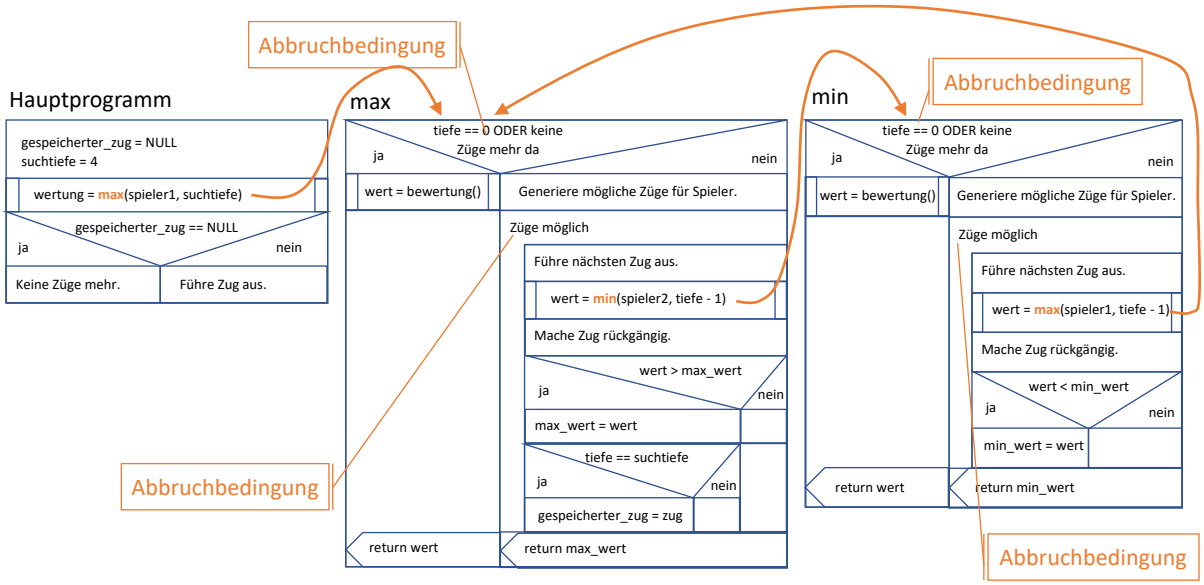


[14] Abbildung nach (wikipedia - Tiefensuche, 2022)
[15] Aus: (Adorf, 2009, S. 9)
[16] Bildquelle: (Nogueira, 2019)

Abbildung 3.3 zeigt die Wirkungsweise des MinMax-Algorithmus. Kreise symbolisieren dabei die Züge des Spielers Max, für den der Algorithmus angewendet wird. Die Quadrate stellen die Züge des Gegenspielers Min dar. Die Werte innerhalb der Symbole repräsentieren den Wert des MinMax-Algorithmus. Die roten Pfeile stehen für den jeweils ausgewählten Zug, die Zahlen an der linken Seite für die Suchtiefe und der blaue Pfeil schließlich für den final ausgewählten Zug. Der Einfachheit halber sind im obigen Spielbaum nur jeweils maximal zwei statt sieben Nachfolgezustände dargestellt.

Der bisher beschriebene Algorithmus untersucht den Spielbaum vollständig. Es handelt sich hierbei um eine Anwendung der Tiefensuche. Da es aber – wie bereits zuvor beschrieben – dabei zu einem sehr großen Spielbaum kommt, wird die Suchtiefe begrenzt und als Abbruchbedingung hinzugefügt. Abbildung 3.4 zeigt den resultierenden MinMax-Algorithmus als Struktogramm.

Abbildung 3.4: Struktogramm des MinMax-Algorithmus



3.2.4 Die Bewertungsfunktion für den Nutzwert

Da der bisher beschriebene Algorithmus gleichermaßen für alle Nullsummenspiele anwendbar ist, kommt der Bewertungsfunktion für den Nutzwert eine besondere Bedeutung zu. Sie liefert eine Zahl, die ausdrücken soll, wie gut die Stellung eines Spiels für den jeweiligen Spieler ist.

Beim Spiel „Vier gewinnt“ kommt es darauf an, vier zusammenhängende Steine zu erreichen. Neben der Untersuchung auf eine Gewinnstellung soll jetzt vereinfachend die

Anzahl von Einern, Zweiern und Dreiern in einer Zeile untersucht werden, da diese mithelfen, eine Gewinnposition zu erreichen. Je mehr Steine hintereinander liegen, um so stärker sollen diese Situationen in die Nutzwertberechnung eingehen.[17]

$$wert_s = \sum_{i=1}^{3} gewicht_i * anzahl_i$$

$$wert = wert_{s2} - wert_{S1}$$

Für die Gewichte wurde in der konkreten Implementierung auf die Vorschläge aus (Hilbig, 2014) zurückgegriffen. Die konkrete Funktion wird im Anhang 6.5.2 näher beschrieben.

3.3 Möglichkeiten zur Verbesserung des Ergebnisses

3.3.1 Schwächen der gegenwärtigen Lösung

Ein so implementierter Algorithmus erlaubt bereits den Einsatz als Computergegner bei „Vier gewinnt“. Allerdings weist er auch noch einige Schwächen auf. Der Computer macht immer noch Züge, die nicht sinnvoll erscheinen; insbesondere bei aktuell bevorstehenden Gewinnsituationen scheint er lange Berechnungen durchzuführen, anstatt gleich den sinnvollen Zug zu machen. Bei der Suche im gesamten Spielbaum erscheint es wahrscheinlich, dass dabei viele Schritte durchgeführt werden, die nichts zum Ergebnis beitragen. Außerdem lässt der Computer sehr oft Steine auf der linken Seite fallen, was aus strategischer Sicht nicht besonders sinnvoll ist.

3.3.2 Heuristiken zur Abkürzung der Suche

Es ist einsichtig, dass eine Suche im Spielbaum bei bestimmten Situationen nicht notwendig ist und deswegen gleich darauf verzichtet werden kann. Wenn der nächste Zug einen eigenen Gewinn ermöglicht, soll dieser Zug ausgeführt werden. Wenn hingegen der nächste Zug einen Gewinn des Gegners ermöglichen würde, soll dieser verhindert werden.

3.3.3 Anpassung der Untersuchungsreihenfolge bei der Tiefensuche

Es ist offensichtlich, dass Spielstellungen mit Steinen in der Mitte potenziell mehr Möglichkeiten bereithalten, um Viererketten zu bilden, als wenn die Steine am Rand stünden. Deswegen sollten zuerst die Züge untersucht werden, die in der Mitte möglich sind. Von

[17] Vorschläge für derartige Nutzwertfunktionen finden sich u.a. in (Krusenotto, 2016, S. 133) und (Hilbig, 2014, S. 8).

den möglichen Zügen von links bis rechts sollen demnach zunächst die in der Mitte untersucht werden. Im Ergebnis werden mit einer größeren Wahrscheinlichkeit die Steine in die Mitte anstatt an den Rand gesetzt.[18]

3.3.4 Alpha-Beta-Suche

Intuitiv ist klar, dass man bei einer vollständigen Untersuchung aller Spielstellungen auch sehr viele Stellungen untersuchen muss, die keinen positiven Beitrag zum Endergebnis leisten. Die sogenannte Alpha-Beta-Suche klammert dabei die Knoten aus, die das Endergebnis nicht beeinflussen: α und β bilden ein Intervall, das sich von oben nach unten immer weiter zu zieht. α und β beschreiben ein „Worst Case Szenario“ für beide Spieler. Werte, die außerhalb dieses Intervalls liegen, brauchen dann nicht mehr betrachtet werden.[19]

Abbildung 6.6, Abbildung 6.7 und Abbildung 6.8 in Anhang 6.5.3 zeigen den angepassten MinMax-Algorithmus als Struktogramm. Außerdem ist dort eine schrittweise Beschreibung des Algorithmus zu finden.

4 Umsetzung in Python

Nachdem nun die grundlegenden Algorithmen für das „Vier gewinnt“-Spiel dargestellt und schrittweise verfeinert wurden, folgt nun eine Beschreibung ausgewählter Aspekte der Implementierung in Python.

4.1 Projektstruktur und grundlegende Festlegungen

Es wird erwartet, dass – angesichts der Komplexität der Algorithmen und der erwarteten Lösung – nicht sämtliche Funktionalität in einem Programm untergebracht werden sollte. Die gewählte Struktur wird im Anhang 6.1 näher dargestellt.

Weiterhin wurden folgende Festlegungen für das Projekt getroffen: Das Spiel soll mit einer grafischen Oberfläche dargestellt werden. Die Bedienung soll dabei mit der Maus erfolgen können. Die Implementierung soll rein unter Nutzung prozeduraler Programmierung und nicht unter Einsatz von Objektorientierung erfolgen (Vorgabe durch die Aufgabenstellung). Eigene Funktions- und Variablennamen werden, ebenso wie die Kommentare, auf Deutsch ausgeführt, um das Verständnis und die Lesbarkeit für Anfänger (wie die Autorin und ihre Mitschülerinnen und Mitschüler) zu erleichtern.

---

[18] Sie dazu auch: (Krusenotto, 2016, S. 142)
[19] Aus: (Krusenotto, 2016, S. 141)

4.2 Umsetzung des Spielfeldes

Das Spielfeld für „Vier gewinnt“ beträgt sechs mal sieben Felder. Naheliegend ist daher die Abbildung als Matrix. Da Python von vornherein keine Arrays unterstützt und die Handhabung von Listen in Listen als unhandlich erscheint, wurde auf die Implementierung von Arrays im bekannten Python-Paket numpy zurückgegriffen.[20]

Damit können Arrays benutzt werden, bei denen die einzelnen Felder direkt in der Form feld[zeile, spalte] angesprochen werden können.[21]

Die Abmessungen des Spielfeldes werden, wie auch andere Konstanten, in einer separaten Datei mit zentral definierten Konstanten konstanten.py festgelegt. Das bringt den Vorteil, dass solche Werte leicht an einer zentralen Stelle änderbar sind. Außerdem kann man bei konsequenter Auslegung damit leicht das Spiel anpassen: Aus „Vier gewinnt“ würde beispielsweise ein „Fünf gewinnt“ mit passendem Spielfeld.

4.3 Spielprogrammierung mit pygame

Auf der Suche nach einer Möglichkeit der graphischen Darstellung – am besten mit der Möglichkeit zur Interaktion mit einem Nutzer – wurde das Paket pygame gefunden. Dabei handelt es sich um eine Bibliothek zur Spieleprogrammierung, die Module zum Abspielen und Steuern von Grafik und Sound sowie zum Abfragen von Eingabegeräten (Tastatur, Maus, Joystick) enthält.[22]

Der grundsätzliche Ablauf ist dabei wie folgt: Nach der Initialisierung des Fensters werden in einer Endlosschleife zunächst Ereignisse abgefragt (z.B. Eingaben über die Tastatur oder Maus). Abhängig davon wird darauf reagiert, beispielsweise wird die Logik des Spielschritts ausgeführt. Danach wird das Fenster mit angepasstem Inhalt erneut ausgegeben. Das passiert so lange, bis als Ergebnis eines Ereignisses die Schleife verlassen wird.[23] Abbildung 6.2 in Anhang 6.2.1.2 zeigt den Algorithmus als Struktogramm.

---

[20] Dabei handelt es sich um ein umfangreiches Paket für wissenschaftliches Rechnen in Python. Siehe: (numpy.org, 2022).

[21] Eine alternative, aber schwerer lesbare, Umsetzung hätte in der Nutzung einer Liste bestanden, wo alle Zeilen hintereinander in eine eindimensionale Struktur eingeordnet würden und jedes Mal eine Umrechnung von zweidimensionalen Koordinaten in den Listenindex bzw. umgekehrt hätte erfolgen müssen (durch: `index = breite * zeilennummer + spaltennummer)`. Beispielsweise wird in (Krusenotto, 2016, S. 128 ff.) eine solche Implementierung – wenn auch in LISP – gewählt.

[22] Siehe: (pygame.org, 2022).

[23] Siehe: (Pratzner, 2022). Ein analoger Ansatz wird z.B. auch bei Processing 3 gewählt, einer in der Lehre bekannten, auf Java basierenden Umgebung. Siehe: (Processing Foundation, 2022).

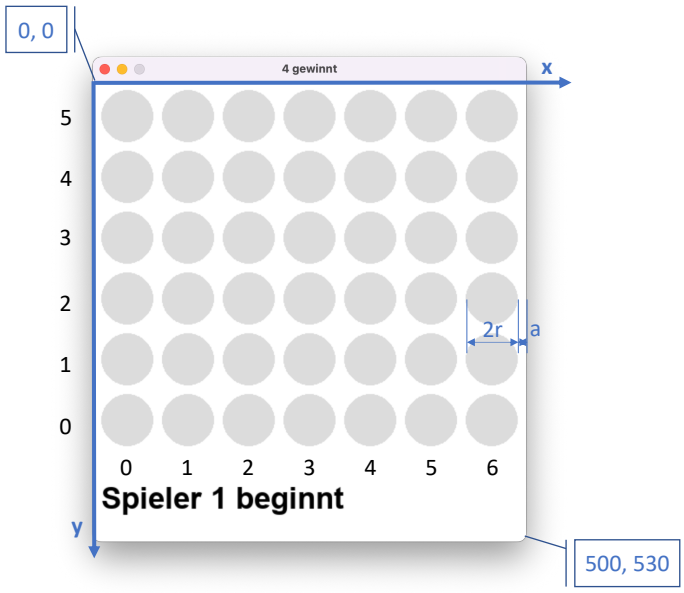
4.4 Das Spielfenster

Das Spielfenster soll aus einer Anordnung von Kreisen bestehen, die passend eingefärbt sind (grau, wenn nicht besetzt; rot und blau für die Darstellung der Spielsteine).

Dabei sollen die Kreise einen bestimmten Radius aufweisen und sowohl zum nächsten Kreis und zu den Rändern einen definierten Abstand einhalten. Außerdem soll am unteren Ende des Fensters Platz für eine Nachrichtenausgabe gelassen werden. Das Programm soll aus Klicks mit der Maus reagieren können. Ein Klick mit der Maus im Bereich einer Spalte des Spielfeldes soll dabei als Aufforderung gewertet werden, einen Spielstein in dieser Spalte fallen zu lassen.

Abbildung 4.1 zeigt das implementierte Spielfenster und die entsprechenden Koordinaten sowie die Berechnung der Maße und außerdem die Ermittlung der Spalte aus Position der Maus.

Abbildung 4.1: Spielfenster mit Koordinaten



$$b = s(2r + a) + a$$

$$h = z(2r + a) + a + h_s$$

$$i_s = \frac{x - r - a}{2r + a}$$

a…Abstand
b…Breite
h…Höhe
h_s…Höhe der Statuszeile
i_s…Spaltenindex
r…Radius
s…Spaltenanzahl
x…x-Position
z…Zeilenanzahl

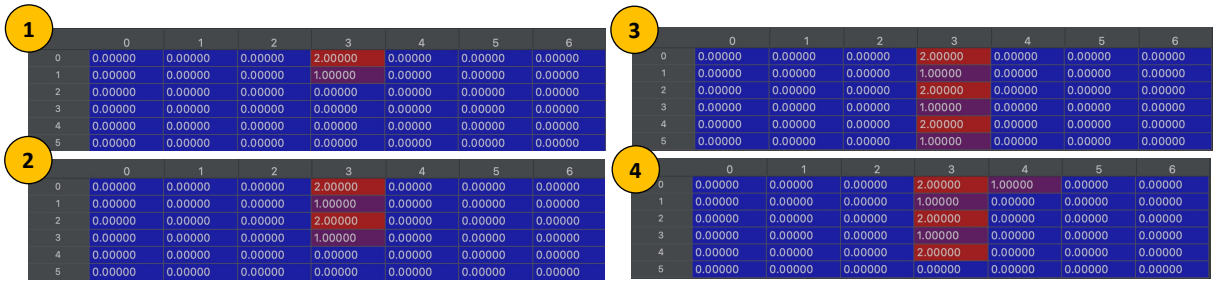
4.5 Ausführung des MinMax-Algorithmus

Wie wir aus Abschnitt 3.2 bereits wissen, wird der MinMax-Algorithmus rekursiv ausgeführt, in dem wechselseitig die Funktionen für Max und Min aufgerufen werden.

Dabei werden schrittweise das Spielfeld aufgebaut (Funktion fuehre_naechsten_zug_aus) und die Spielposition untersucht. Danach wird das Spielfeld schrittweise wieder abgebaut (Funktion nehme_zug_zurueck). Der Algorithmus bricht ab, wenn die definierte Suchtiefe erreicht wurde bzw. keine Züge mehr möglich sind.

Abbildung 4.2 zeigt ausschnittsweise den schrittweisen Aufbau der Datenstruktur für das Spielfeld im Verlauf des MinMax-Algorithmus (Bildschirmfotos der Datenstruktur im Debugger).

Abbildung 4.2: Schrittweiser Aufbau des Spielfeldes beim MinMax-Algorithmus



Im Einzelnen muss der Algorithmus sehr häufig durchlaufen werden. Die folgenden Daten wurden in Abhängigkeit von der Suchtiefe für ein nahezu leeres Spielfeld – und damit für besonders viele zu untersuchende Positionen – ermittelt (Der Spieler macht den ersten Zug in Spalte 3, der Computer soll den nächsten Zug machen).

Tabelle 4.1: Aufrufe und Laufzeiten in Abhängigkeit der Suchtiefe

Suchtiefe	Zug in Spalte	Max-Wert	Aufrufe Max	Aufrufe Min	Stellungen	Laufzeit [s]
4	3	1	111	62	194	0,01
5	4	1	389	1793	3356	0,1
6	3	1	1375	563	2162	0,07
7	3	15	7424	37977	69781	2,28
8	3	1	19243	5875	20219	1,05
9	5	17	122707	580310	1070254	39,76
10	3	1	353667	85024	585348	22,8

Es ist zu sehen, dass der Algorithmus nur bis zu einer bestimmten Suchtiefe (hier 8) sinnvoll anwendbar ist, da sonst die Wartezeiten für den Spieler zu lange ausfallen.

5 Zusammenfassung

5.1 Bewertung der Lösung

Die entstandene Lösung ermöglicht das Spiel „Vier gewinnt“ mit ansprechender grafischer Oberfläche auf einem Computer, sowohl von zwei menschlichen Spielern gegeneinander als auch gegen den Computer. Der implementierte Algorithmus bietet genug

Spielstärke, dass ein Sieg des Einzelspielers gegen den Computer nicht trivial ist und dabei trotzdem die Berechnungszeiten akzeptabel bleiben.

5.2 Ideen für weitere Verbesserungen

Die implementierte Lösung könnte noch in diversen Aspekten – z.B. Erweiterung der Spielfunktionen und Verbesserung der Laufzeit – erweitert und verbessert werden.

Die Spielfunktionen bilden momentan ein Minimum der denkbaren Funktionalität ab. Erweiterungen sind beispielsweise für die Darstellung der Gewinnstellung, Einstellmöglichkeiten durch den Nutzer, Zurücknehmen von Zügen und Speichern/Laden des Spielstandes denkbar (siehe Anhang 6.8).

Bisher wird die praktisch erträgliche Suchtiefe durch die Länge der Laufzeit des MinMax-Algorithmus bestimmt. Der Algorithmus mit seiner Anwendung auf einem Spielbaum erscheint geeignet, dass verschiedene Zweige des Baumes parallel untersucht werden, um mehrere Prozessorkerne des Computers mit dem Ziel der Verbesserung der Laufzeit auszunutzen. Es existieren Untersuchungen für solche Implementierungen.[24]

5.3 Lernpotenzial für weitere Projekte

Für die Autorin stellte die Arbeit an „Vier gewinnt“ das erste größere Software-Projekt dar. Folgende Aspekte wurden als Lernpotenzial identifiziert, die bei Folgeprojekten berücksichtigt werden sollten.

Die grundlegenden Programmfunktionen müssen vor dem Programmieren überlegt werden. Die verwendeten Algorithmen sollten – je nach Komplexität – vor dem Programmieren überlegt und visualisiert werden. Komplexere Projekte brauchen eine Unterstruktur. Python hat nützliche Bibliotheken für viele Zwecke. Eine Recherche je nach Anwendungsfall kann sehr nützlich sein.

Sprechende Variablennamen, DocStrings und Kommentare helfen, das Programm auch noch später zu verstehen. Die Nutzung von zentral definierten Konstanten anstelle von Literalen (festen Werten im Programmcode) hilft, diese schnell anzupassen. Die Nutzung des Programms wird außerdem flexibler. Die Nutzung von Unterprogrammen hilft, die Übersicht zu behalten. Rekursive Programme sind gut zu lesen, aber schwierig zu debuggen.

---

[24] Ein Ansatz für die Parallelisierung der Abarbeitung findet sich in (Department of Computer Science and Engineering, Univerity of Washington, 2022, S. 6 ff.)

Literatur- und Quellenverzeichnis

Adorf, J. (02. Dezember 2009). *Adversariale Suche für optimales Spiel: Der Minimax-Algorithmus und die Alpha-Beta-Suche.* Von juliusadorf.com: https://www.juliusadorf.com/pub/alphabeta-seminar-paper.pdf abgerufen

Allen, J. D. (2010). *The Complete Book of Connect 4: History, Strategy, Puzzles.* New York: Sterling Publishing Company, Incorporated.

Allis, V. (Oktober 1988). *A Knowledge-based Approach of Connect-Four.* Von uni-trier.de: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf abgerufen

Antonsusi. (27. August 2004). *Alpha-beta Suchbaum.* Von wikimedia.org: https://commons.wikimedia.org/wiki/File:Alpha_beta.svg abgerufen

Department of Computer Science and Engineering, Univerity of Washington. (31. Oktober 2022). *CSE332: Data Structures and Parallelism: Games, Minimax, and Alpha-Beta Pruning.* Von The CSE332 Web: https://courses.cs.washington.edu/courses/cse332/17wi/documents/games.pdf abgerufen

Edelkamp, S., & Kissmann, P. (2008). *Symbolic Classification of General Two-Player Games.* Von uni-saarland.de: https://fai.cs.uni-saarland.de/kissmann/publications/ki08-ggpsolver.pdf abgerufen

Hasbro SA. (2020). *4 gewinnt.* Von hasbro.com: https://instructions.hasbro.com/de-de/all-instructions?search=vier%20gewinnt abgerufen

Hilbig, T. (04. November 2014). *Optimierung und Laufzeitanalyse einer künstlichen Intelligenz für das Spiel Vier gewinnt.* Von hans-riegel-fachpreise.com: https://www.hans-riegel-fachpreise.com/fileadmin/hans-riegel-fachpreise/Module/ausgezeichnete-arbeiten/hans-riegel-fachpreise-seminararbeit-vwa-2015-hilbig.pdf abgerufen

Kjeldsen, T. H. (November 2001). *John von Neumann's Conception of the Minimax Theorem: A Journey Through Different Mathematical Contexts.* Von reserchgate.net: https://www.researchgate.net/publication/225823001_John_von_Neumann%27s_Conception_of_the_Minimax_Theorem_A_Journey_Through_Different_Mathematical_Contexts abgerufen

Krusenotto, P. (2016). *Funktionale Programmierung und Metaprogrammierung.* Wiesbaden: Springer Fachmedien.

Neumann, J. v. (1928). Zur Theorie der Gesellschaftsspiele. In A. Clebsch, & K. Neumann, *Mathematische Annalen, 10. Band.* Berlin: Julius Springer.

Nogueira, N. (11. Februar 2019). *Minimax.* Von wikimedia.org: https://commons.wikimedia.org/w/index.php?curid=2276653 abgerufen

numpy.org. (06. November 2022). *numpy.org*. Abgerufen am 06. November 2022 von numpy.org: https://numpy.org

Pratzner, A. (07. Februar 2022). *Pygame Tutorial*. Abgerufen am 06. November 2022 von python-lernen.de: https://www.python-lernen.de/pygame-tutorial.htm

Processing Foundation. (06. November 2022). *Processing*. Abgerufen am 06. November 2022 von processing.org: https://processing.org

pygame.org. (06. November 2022). *pygame.org*. Abgerufen am 06. November 2022 von pygame.org: https://www.pygame.org/wiki/about

Ruile, B., Weiß, B., Ditsche, M., & Schmidt, N. (2009). *Projektarbeit Vier Gewinnt.* Von uni-muenchen.de: https://www.mathematik.uni-muenchen.de/~spielth/artikel/VierGewinnt.pdf abgerufen

wikipedia - Alpha-Beta-Suche. (30. Oktober 2022). *Alpha-Beta-Suche.* Von wikipedia.de: https://de.wikipedia.org/wiki/Alpha-Beta-Suche abgerufen

wikipedia - Breitensuche. (06. August 2022). *Breitensuche.* Von wikipedia.de: https://de.wikipedia.org/wiki/Breitensuche abgerufen

wikipedia - rekursive Programmierung. (29. März 2022). *Rekursive Programmierung.* Von wikipedia.de: https://de.wikipedia.org/wiki/Rekursive_Programmierung abgerufen

wikipedia - Tiefensuche. (29. Juli 2022). *Tiefensuche.* Von wikipedia.de: https://de.wikipedia.org/wiki/Tiefensuche abgerufen

wikipedia - vier gewinnt. (11. März 2022). *vier gewinnt*. Von wikipedia.de: https://de.wikipedia.org/wiki/Vier_gewinnt abgerufen

6 Anhang

6.1 Projektstruktur

Tabelle 6.1: Projektstruktur

Funktion	Erklärung
main.py	Enthält die “main loop” für das Spiel und die grundlegende Spiellogik.
ki.py	Enthält die Funktionen für den Computergegner.
konstanten.py	Enthält die im Projekt verwendeten Konstanten.
spiel.py	Enthält Logik zu Spielern (und Logik zum Berechnen von Zügen).
spielfeld.py	Enthält die Funktionen zum Spielfeld (Erzeugen, Ausgabe, Gewinnposition, mögliche Züge).
performance.py	Ein Objekt, um Performancedaten zu speichern. Hier wurde entgegen der Vorgabe Objektorientierung eingesetzt. Da dies aber nicht zum eigentlichen Algorithmus gehört und diese Lösung viel eleganter ist, wurde darauf zurückgegriffen.

6.2 Hauptprogramm – main.py

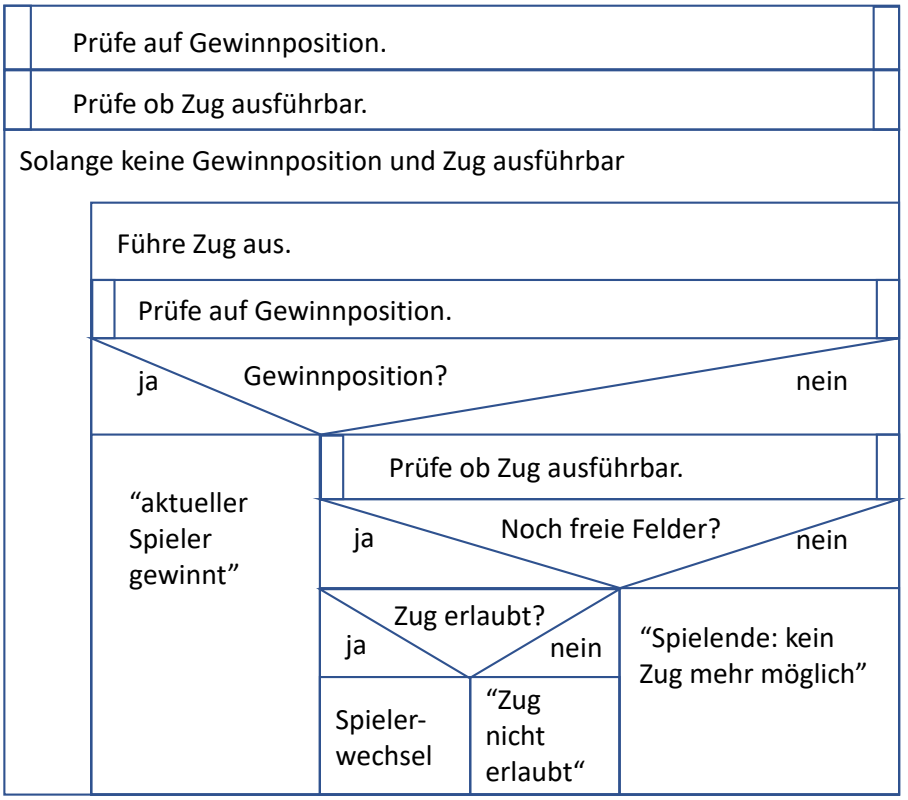
6.2.1 Ausgewählte Algorithmen

6.2.1.1 Grundlegender Spielalgorithmus

Solange das Spiel weitergespielt werden kann, führe folgende Schritte aus:

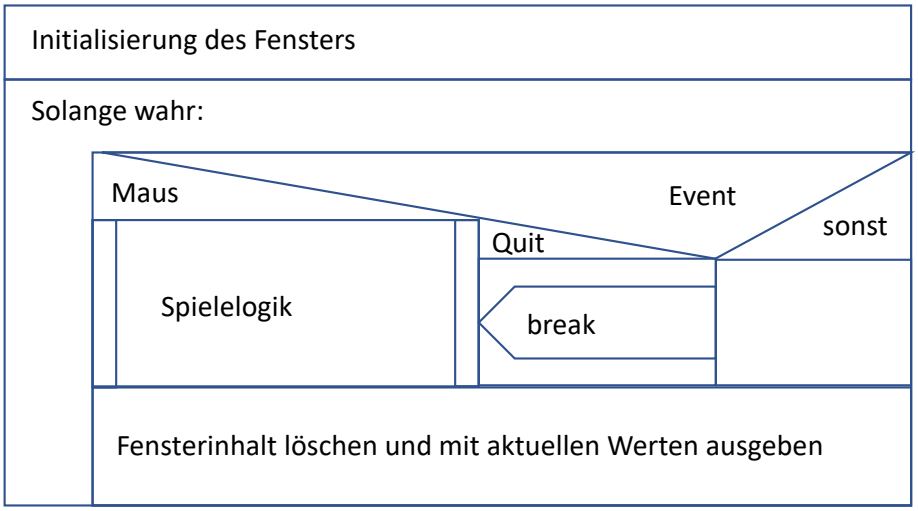
- *Führe den Zug aus.*
- *Ist durch den Zug eine Gewinnposition erreicht?*
    - *Ja: Das Spiel ist zu Ende. Der aktuelle Spieler gewinnt.*
    - *Nein: Ist ein weiterer Zug möglich?*
        - *Ja: Ist der Zug erlaubt?*
            - *Ja: Spielerwechsel*
            - *Nein: Nachricht: Der Zug ist nicht erlaubt.*
        - *Nein: Nachricht: Das Spiel ist zu Ende. Es ist kein Zug mehr möglich.*

Abbildung 6.1: Grundlegender Spielalgorithmus



6.2.1.2 Grundlegender Algorithmus zur Spielsteuerung

Abbildung 6.2: Grundlegender Algorithmus zur Spielsteuerung



6.2.2 Programmcode

```
#! /usr/bin/env python

"""
Hauptprogramm für ein "4 gewinnt"-Spiel
"""
__author__ = "Henriette Schulz"
__version__ = "1.0.0"
__email__ = "schulz.henriette.0106@gmail.com"
__status__ = "Prototype"
```

```
import spielfeld as sf
import konstanten as ko
import spiel
import ki

import pygame
from pygame.locals import *
import sys

pygame.init()

if __name__ == '__main__':
    titel = str(ko.GEWINNFELDER) + " gewinnt"
    spielfeld = sf.erzeuge_leeres_spielfeld(ko.ZEILEN, ko.SPALTEN)

    # Definieren und Öffnen eines neuen Fensters
    fenster = pygame.display.set_mode((ko.W, ko.H))
    pygame.display.set_caption(titel)
    clock = pygame.time.Clock()

    # setze Spielvariablen und initiale Nachricht
    spieler = spiel.spieler2
    nachricht = spiel.liefere_name(spieler) + " beginnt"
    nachricht2 = "Klicke für nächsten Zug."

    game_over = False
    erster_zug = True
    gewinn_erreicht = False
    zuege_vorbei = False
    zug_erlaubt = True

    # Schleife Hauptprogramm
    while True:
        # Überprüfen, ob Nutzer eine Aktion durchgeführt hat
        for event in pygame.event.get():
            # Beenden bei [ESC] oder [X]
            if event.type == QUIT or (event.type == KEYDOWN and event.key ==
K_ESCAPE):
                pygame.quit()
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONUP and not game_over:
                # Spiellogik
                spielername = spiel.liefere_name(spieler)
                if spielername == "Computer" and ko.KI_MODUS:
                    # Spielfeld löschen
                    fenster.fill(ko.WEISS)
                    # Spielfeld/figuren zeichnen
                    sf.plotte_spielfeld(spielfeld, fenster, nachricht, "Zug
wird berechnet.")
                    # Fenster aktualisieren
                    pygame.display.flip()
                    clock.tick(ko.FPS)

                    zug = ki.schlage_naechsten_zug_vor(spielfeld, spieler,
erster_zug)
                    erster_zug = False
                    gewinn_erreicht, zug_erlaubt = spiel.mache_zug(spielfeld,
spieler, zug["spalte"])
                    zuege_vorbei = not spiel.ist_zug_moeglich(spielfeld)
                    game_over = gewinn_erreicht or zuege_vorbei
                else:
```

```
                    pos = pygame.mouse.get_pos()
                    spalte = sf.ermittle_maus_spalte(pos)
                    gewinn_erreicht, zug_erlaubt = spiel.mache_zug(spielfeld,
spieler, spalte)
                    zuege_vorbei = not spiel.ist_zug_moeglich(spielfeld)
                    game_over = gewinn_erreicht or zuege_vorbei

                if not zug_erlaubt:
                    nachricht = spiel.liefere_name(spieler) + ": Zug nicht
erlaubt"
                elif not game_over:
                    nachricht = "nächster Zug für " + spiel.lie-
fere_name(spiel.liefere_anderen_spieler(spieler))
                else:
                    nachricht2 = "ESC für Spielende"
                    if zuege_vorbei and gewinn_erreicht:
                        nachricht = spiel.liefere_name(spieler) + " gewinnt"
                    elif zuege_vorbei:
                        nachricht = "keine Züge mehr möglich"
                    else:
                        nachricht = spiel.liefere_name(spieler) + " gewinnt"

                # Spieler wechseln
                if zug_erlaubt:
                    spieler = spiel.liefere_anderen_spieler(spieler)

        # Spielfeld löschen
        fenster.fill(ko.WEISS)

        # Spielfeld/figuren zeichnen
        sf.plotte_spielfeld(spielfeld, fenster, nachricht, nachricht2)

        # Fenster aktualisieren
        pygame.display.flip()
        clock.tick(ko.FPS)
```

6.3 Funktionen für das Spielfeld – spielfeld.py

6.3.1 Implementierte Funktionen

Die folgende Tabelle listet alle in spielfeld.py implementierten Funktionen auf. Die Parameter sind in den DocStrings des beigefügten Programmcodes erklärt.

Tabelle 6.2: Implementierte Funktionen für das Spielfeld

Funktion	Erklärung
erzeuge_leeres_spielfeld	Erzeugt ein leeres (mit 0 gefülltes) Array als Spielfeld.
plotte_spielfeld	Zeichnet das Spielfeld und die Spielfiguren. Gibt eine Nachricht aus.

liefere_erste_freie_zeile	Liefert die erste unbesetzte Zeile für eine gegebene Spalte.
mache_zug	Führt einen Zug aus.
liefere_freie_spalten	Ermittelt die Spalten und die zugehörigen ersten freien Zeilen, auf denen Züge möglich sind.
ermittle_maus_spalte	Ermittelt die Spalte aus der gegebenen x, y-Position.
koordinaten_im_feld	Prüft, ob die Indizes auf dem Spielfeld sind.
pruefe_spalte_fuer_gewinn	Prüft in gegebener Spalte, ob eine Gewinnposition für den gegebenen Spieler vorliegt.
pruefe_zeile_fuer_gewinn	Prüft in gegebener Zeile, ob eine Gewinnposition für den gegebenen Spieler vorliegt.
pruefe_diagonale_rechts_oben_links_unten_fuer_gewinn	Prüft beabsichtigten Zug in der betreffenden Diagonale von rechts oben nach links unten, ob eine Gewinnposition für den gegebenen Spieler vorliegt.
pruefe_diagonale_rechts_unten_links_oben_fuer_gewinn	Prüft beabsichtigten Zug in der betreffenden Diagonale von rechts unten nach links oben, ob eine Gewinnposition für den gegebenen Spieler vorliegt.
ist_zug_gewinn	Prüft, ob der beabsichtigte Zug für den gegebenen Spieler zu einem Gewinn führt.
nehme_zug_zurueck	Setzt Feld für gegebenen Zug zurück auf 0 und nimmt damit den Zug zurück.
liefere_benutzte_felder	Liefert eine Liste der benutzten Felder.

6.3.2 Ausgewählte Algorithmen

6.3.2.1 Ermittlung der möglichen Züge

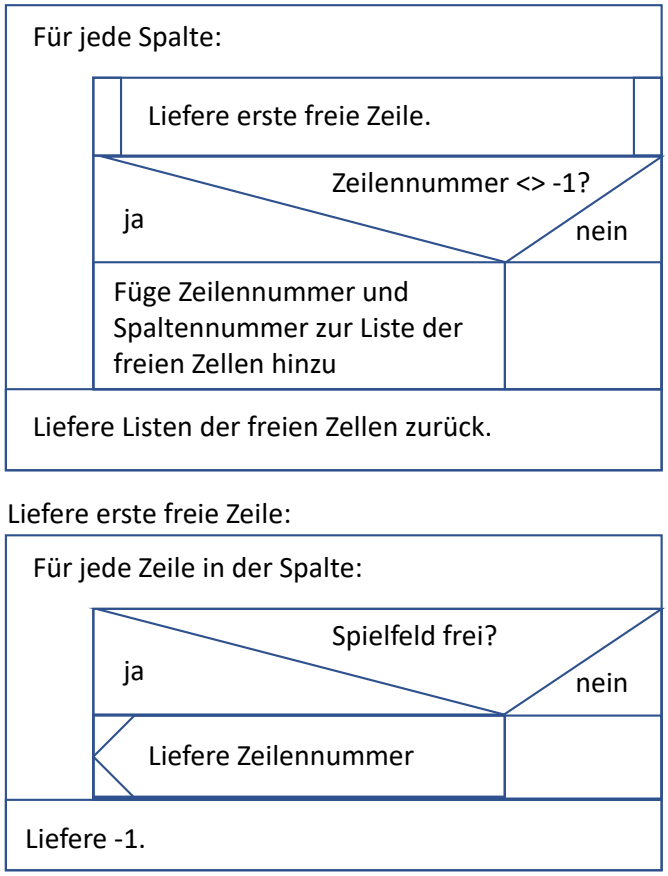
Für jede Spalte führe folgende Schritte aus:

- *Liefere die erste freie Zelle*

- *Existiert eine freie Zelle?*
    - *Ja: Füge die Zeilennummer und Spaltennummer zur Liste der freien Zellen hinzu.*

Gib die Liste der freien Zellen zurück.

Abbildung 6.3: Algorithmus zur Ermittlung der möglichen Züge



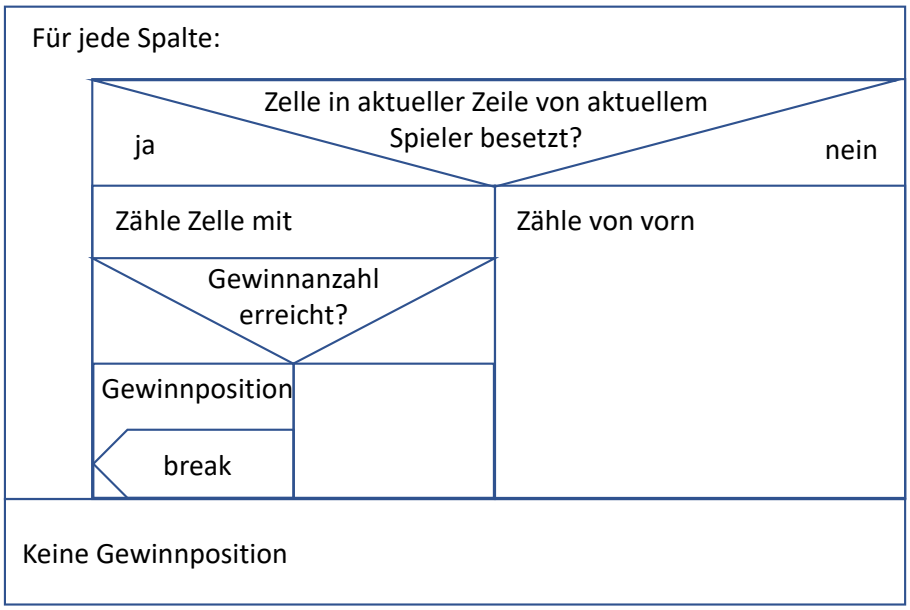
6.3.2.2 Prüfung auf Gewinnposition in einer Zeile

Für jede Spalte innerhalb der Zeile führe aus:

- *Ist die Zelle der Zeile vom Spieler besetzt?*
    - *Ja:*
        - *Zähle die Zelle mit.*
        - *Wurde die Gewinnanzahl erreicht?*
            - *Ja: Gewinnposition, Abbruch der Schleife.*
    - *Nein: Zähle von vorn.*

Wenn diese Stelle nach Ende der Schleife erreicht wird, wurde keine Gewinnposition erkannt.

Abbildung 6.4: Algorithmus zum Erkennen der Gewinnposition in einer Zeile



6.3.3 Programmcode

```
#! /usr/bin/env python

"""
Funktionen für das Spielfeld für ein "4 gewinnt"-Spiel
"""
__author__ = "Henriette Schulz"
__version__ = "1.0.0"
__email__ = "schulz.henriette.0106@gmail.com"
__status__ = "Prototype"

import numpy
import konstanten as ko

import pygame
from pygame.locals import *


def erzeuge_leeres_spielfeld(zeilen, spalten):
    """
    Erzeugt ein leeres (mit 0 gefülltes) Array als Spielfeld.
    :param zeilen: Anzahl der Zeilen
    :param spalten: Anzahl der Spalten
    :return: zweidimensionales Array als Spielfeld
    """
    return numpy.zeros((zeilen, spalten))


def plotte_spielfeld(spielfeld, fenster, nachricht1, nachricht2):
    """
    Zeichnet das Spielfeld und die Spielfiguren. Gibt eine Nachricht aus.
    :param spielfeld: Array mit den Spielinformationen
    :param fenster: Fenster zum Zeichnen
    :param nachricht1: Text für Statusnachricht groß
    :param nachricht2: Text für Statusnachricht klein
    :return: nichts
    """
    farbe = ko.GRAU
```

```
    schrift = pygame.font.SysFont('Arial', 30, True, False)
    schrift_klein = pygame.font.SysFont('Arial', 20, True, False)
    text1 = schrift.render(nachricht1, True, ko.SCHWARZ)
    text2 = schrift_klein.render(nachricht2, True, ko.SCHWARZ)
    fenster.blit(text1, [ko.ABSTAND, ko.H - 80])
    fenster.blit(text2, [ko.ABSTAND, ko.H - 40])

    for z in range(ko.ZEILEN):
        zeile = ko.ZEILEN - z - 1
        for spalte in range(ko.SPALTEN):
            if spielfeld[z, spalte] == 0:
                farbe = ko.GRAU
            if spielfeld[z, spalte] == ko.SPIELER1:
                farbe = ko.ROT
            if spielfeld[z, spalte] == ko.SPIELER2:
                farbe = ko.BLAU

            x = ko.RADIUS + (spalte + 1) * ko.ABSTAND + spalte * 2 * ko.RA-
DIUS
            y = ko.RADIUS + (zeile + 1) * ko.ABSTAND + zeile * 2 * ko.RADIUS

            pygame.draw.circle(fenster, farbe, (x, y), ko.RADIUS)


def liefere_erste_freie_zeile(spielfeld, spalte):
    """
    Liefert die erste unbesetzte Zeile für eine gegebene Spalte.
    :param spielfeld: Array mit den Spielinformationen
    :param spalte: zu untersuchende Spalte
    :return: Index der ersten freien Zeile; -1, wenn keine freie Zeile gefun-
den
    """
    if spalte >= ko.SPALTEN:
        return -1

    for i in range(ko.ZEILEN):
        if spielfeld[i, spalte] == 0:
            return i

    return -1


def mache_zug(spielfeld, spieler, spalte):
    """
    Führt einen Zug aus.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: aktueller Spieler
    :param spalte: Index der Spalte, wo der Zug ausgeführt wird
    :return: True, wenn der Zug zum Gewinn führt; sonst False; True, wenn Zug
möglich, sonst False
    """
    zeile = liefere_erste_freie_zeile(spielfeld, spalte)
    zug_erlaubt = True
    if zeile >= 0:
        spielfeld[zeile, spalte] = spieler
    else: zug_erlaubt = False

    return ist_zug_gewinn(spielfeld, spieler, spalte, zeile), zug_erlaubt


def liefere_freie_spalten(spielfeld):
    """
```

```
    Ermittelt die Spalten und die zugehörigen ersten freien Zeilen, auf denen
Züge möglich sind.
    :param spielfeld: Array mit Spielinformationen
    :return: Liste von Dictionaries {"spieler", "zeile", "spalte"}
    """
    freie_spalten = []
    for spalte in range(ko.SPALTEN):
        m = {"spieler": 0,
             "zeile": liefere_erste_freie_zeile(spielfeld, spalte),
             "spalte": spalte}

        if m["zeile"] > -1:
            freie_spalten.append(m)  # there are free fields in this column
    return freie_spalten


def ermittle_maus_spalte(pos):
    """
    Ermittelt die Spalte aus der gegebenen x, y-Position
    :param pos: Position (x, y)
    :return: Index der Spalte, auf die geklickt wurde
    """
    x, y = pos
    return int(((x - ko.RADIUS - ko.ABSTAND) / (ko.ABSTAND + 2 * ko.RADIUS))
+ 0.5)


def koordinaten_im_feld(spalte, zeile):
    """
    Prüft, ob die Indizes auf dem Spielfeld sind.
    :param spalte: Spaltenindex
    :param zeile: Zeilenindex
    :return: True, wenn auf dem Spielfeld; sonst False
    """
    if (spalte >= 0) and (spalte < ko.SPALTEN) and (zeile >= 0) and (zeile <
ko.ZEILEN):
        return True
    else:
        return False


def pruefe_spalte_fuer_gewinn(spielfeld, spieler, spalte, zeile):
    """
    Prüft in gegebener Spalte, ob Gewinnposition für Spieler vorliegt
    :param spielfeld: Array mit Spielinformationen
    :param spieler: Spieler
    :param spalte: zu untersuchende Spalte
    :param zeile: Zeile, auf die nächster Zug fällt
    :return: True für Gewinnposition, sonst False
    """
    gesamt = 0
    for z in range(ko.ZEILEN):
        if (spielfeld[z, spalte] == spieler) or (z == zeile):
            # Das Feld ist von diesem Spieler besetzt.
            # Zähle Feld mit.
            gesamt += 1
            if gesamt == ko.GEWINNFELDER:
                # Anzahl der Gewinnfelder ist erreicht
                return True
        else:
            # Das nächste Feld ist von anderem Spieler besetzt.
            # Fange wieder von vorn an zu zählen.
```

```
            gesamt = 0
    return False


def pruefe_zeile_fuer_gewinn(spielfeld, spieler, spalte, zeile):
    """
    Prüft in gegebener Zeile, ob Gewinnposition für Spieler vorliegt
    :param spielfeld: Array mit Spielinformationen
    :param spieler: Spieler
    :param spalte: Spalte, auf die nächster Zug fällt
    :param zeile: zu untersuchende Zeile
    :return: True für Gewinnposition, sonst False
    """
    gesamt = 0
    for s in range(ko.SPALTEN):
        if (spielfeld[zeile, s] == spieler) or (s == spalte):
            # Das Feld ist von diesem Spieler besetzt.
            # Zähle das Feld mit.
            gesamt += 1
            if gesamt == ko.GEWINNFELDER:
                # Anzahl der Gewinnfelder ist erreicht
                return True
        else:
            # Das nächste Feld ist von anderem Spieler besetzt.
            # Fange wieder von vorn an zu zählen.
            gesamt = 0
    return False


def pruefe_diagonale_rechts_oben_links_unten_fuer_gewinn(spielfeld, spieler,
spalte, zeile):
    """
    Prüft beabsichtigten Zug in Diagonale von rechts oben nach links unten,
ob Gewinnposition für Spieler vorliegt
    :param spielfeld: Array mit Spielinformationen
    :param spieler: Spieler
    :param spalte: Spalte, auf die nächster Zug fällt
    :param zeile: Zeile, auf die nächster Zug fällt
    :return: True für Gewinnposition, sonst False
    """
    gesamt = 0
    # lege Startkoordinaten fest
    z = zeile
    s = spalte

    # suche nach dem Endpunkt in oberer rechter Ecke
    # iteriere nach oben rechts
    while koordinaten_im_feld(s, z):
        z -= 1
        s += 1

    z += 1
    s -= 1

    # iteriere von dort aus nach unten links
    while koordinaten_im_feld(s, z):
        if (spielfeld[z, s] == spieler) or ((z == zeile) and (s == spalte)):
            # Das Feld ist von diesem Spieler besetzt.
            # Zähle das Feld mit.
            gesamt += 1
            if gesamt == ko.GEWINNFELDER:
                # Anzahl der Gewinnfelder ist erreicht
```

```
                return True
        else:
            # Das nächste Feld ist von anderem Spieler besetzt.
            # Fange wieder von vorn an zu zählen.
            gesamt = 0

        z += 1
        s -= 1

    return False


def pruefe_diagonale_rechts_unten_links_oben_fuer_gewinn(spielfeld, spieler,
spalte, zeile):
    """
    Prüft beabsichtigten Zug in Diagonale von rechts unten nach links oben,
ob Gewinnposition für Spieler vorliegt
    :param spielfeld: Array mit Spielinformationen
    :param spieler: Spieler
    :param spalte: Spalte, auf die nächster Zug fällt
    :param zeile: Zeile, auf die nächster Zug fällt
    :return: True für Gewinnposition, sonst False
    """
    gesamt = 0
    # lege Startkoordinaten fest
    z = zeile
    s = spalte

    # suche nach dem Endpunkt in oberer linker Ecke
    # iteriere nach oben links
    while koordinaten_im_feld(s, z):
        z -= 1
        s -= 1

    z += 1
    s += 1

    # iteriere von dort aus nach unten rechts
    while koordinaten_im_feld(s, z):
        if (spielfeld[z, s] == spieler) or ((z == zeile) and (s == spalte)):
            # Das Feld ist von diesem Spieler besetzt.
            # Zähle das Feld mit.
            gesamt += 1
            if gesamt == ko.GEWINNFELDER:
                # Anzahl der Gewinnfelder ist erreicht
                return True
        else:
            # Das nächste Feld ist von anderem Spieler besetzt.
            # Fange wieder von vorn an zu zählen.
            gesamt = 0

        z += 1
        s += 1

    return False


def ist_zug_gewinn(spielfeld, spieler, spalte, zeile):
    """
    Prüft, ob Zug für Spieler zu Gewinn führt.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: Spieler
```

```
    :param spalte: Spalte, auf die nächster Zug fällt
    :param zeile: Zeile, auf die nächster Zug fällt
    :return: True für Gewinnposition, sonst False
    """
    return pruefe_spalte_fuer_gewinn(spielfeld, spieler, spalte, zeile) or \
           pruefe_zeile_fuer_gewinn(spielfeld, spieler, spalte, zeile) or \
           pruefe_diagonale_rechts_oben_links_unten_fuer_gewinn(spielfeld,
spieler, spalte, zeile) or \
           pruefe_diagonale_rechts_unten_links_oben_fuer_gewinn(spielfeld,
spieler, spalte, zeile)


def nehme_zug_zurueck(spielfeld, zug):
    """
    Setzt Feld für gegebenen Zug zurück auf 0 und nimmt damit den Zug zurück.
    :param spielfeld: Array mit Spielinformationen
    :param zug: Spielzug (Dictionary {"spieler", "spalte", "zeile"})
    :return: nichts
    """
    spielfeld[zug["zeile"], zug["spalte"]] = 0
    return


def liefere_benutzte_felder(spielfeld):
    """
    Liefert eine Liste der benutzten Felder.
    :param spielfeld: Array mit Spielinformationen
    :return: Liste von Dictionaries {"spieler", "zeile", "spalte"}
    """
    felder = []
    for spalte in range(ko.SPALTEN):
        for zeile in range(ko.ZEILEN):
            if spielfeld[zeile, spalte] != 0:
                zug = {"spieler": spielfeld[zeile, spalte],
                       "zeile": zeile,
                       "spalte": spalte}
                felder.append(zug)

    return felder
```

6.4 Grundlegende Spielfunktionen – spiel.py

6.4.1 Implementierte Funktionen

Die folgende Tabelle listet alle in spiel.py implementierten Funktionen auf. Die Parameter sind in den DocStrings des beigefügten Programmcodes erklärt.

Tabelle 6.3: Implementierte Funktionen für das Spiel

Funktion	Erklärung
erzeuge_moegliche_zuege	Ermittelt die Spalten und die zugehörigen ersten freien Zeilen, auf denen Züge möglich sind. (Aufruf der zugehörigen Funktion des Spielfeldes.)

liefere_name	Liefert die Bezeichnung des Spielers.
liefere_anderen_spieler	Liefert die ID des anderen Spielers.
ist_zug_moeglich	Prüft, ob noch ein Zug möglich ist.
mache_zug	Führt einen Zug aus. (Aufruf der zugehörigen Funktion des Spielfeldes.)

6.4.2 Programmcode

```
#! /usr/bin/env python

"""
Implementierung der grundsätzlichen Spielfunktionen für ein "4 gewinnt"-Spiel
"""
__author__ = "Henriette Schulz"
__version__ = "1.0.0"
__email__ = "schulz.henriette.0106@gmail.com"
__status__ = "Prototype"

import spielfeld as sf
import konstanten as ko

spieler1 = ko.SPIELER1       # muss größer sein als spieler2 + Anzahl der Ge-
winnfelder
spieler2 = ko.SPIELER2


def erzeuge_moegliche_zuege(spielfeld):
    """
    Ermittelt die Spalten und die zugehörigen ersten freien Zeilen, auf denen
Züge möglich sind.
    :param spielfeld: Array mit Spielinformationen
    :return: Liste von Dictionaries {"spieler", "zeile", "spalte"}
    """
    return sf.liefere_freie_spalten(spielfeld)


def liefere_name(spieler):
    """
    Liefert die Bezeichnung des Spielers
    :param spieler: ID des Spielers
    :return: Text mit dem Namen des Spielers
    """
    if spieler == spieler1:
        return ko.SPIELERNAME1
    else:
        return ko.SPIELERNAME2


def liefere_anderen_spieler(spieler):
    """
    Liefert die ID des anderen Spielers.
    :param spieler: ID des aktuellen Spielers
    :return: ID des anderen Spielers
    """
    if spieler == spieler1:
        return spieler2
```

```
    else:
        return spieler1


def ist_zug_moeglich(spielfeld):
    """
    Prüft, ob noch ein Zug möglich ist.
    :param spielfeld: Array mit Spielinformationen
    :return: True, wenn Zug möglich; sonst False
    """
    if bool(erzeuge_moegliche_zuege(spielfeld)):
        return True
    else:
        return False


def mache_zug(spielfeld, spieler, spalte):
    """
    Führt einen Zug aus.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: aktueller Spieler
    :param spalte: Index der Spalte, wo der Zug ausgeführt wird
    :return: True, wenn der Zug zum Gewinn führt; sonst False; True, wenn Zug
erlaubt ist, sonst False
    """

    zug_gewinn, zug_erlaubt = sf.mache_zug(spielfeld, spieler, spalte)
    return zug_gewinn, zug_erlaubt
```

6.5 Funktionen für „künstliche Intelligenz“

6.5.1 Implementierte Funktionen

Die folgende Tabelle listet alle in ki.py implementierten Funktionen auf. Die Parameter sind in den DocStrings des beigefügten Programmcodes erklärt.

Tabelle 6.4: Implementierte Funktionen für KI-Funktionen

Funktion	Erklärung
bewerte_zelle	Bewertet Zug für den MinMax-Algorithmus.
suche_gewinnposition	Bewertet das gesamte Spielfeld, ob ein Spieler gewonnen hat.
zaehle_anzahl_in_zeile	Zählt die Zahl der Steine in einer Zeile für Spieler - jeweils für 1 bis zur gegebenen Anzahl.
berechne_wert	Berechnet einen gewichteten Wert je nach Anzahl der Spielsteine in einer Zeile.

bewerte_spiel	Bewertet das gesamte Spielfeld, ob ein Spieler gewonnen hat.
fuehre_naechsten_zug_aus	Führt den nächsten Zug aus einer Liste von Zügen aus.
maxx	Stellt die MAX-Methode für den MinMax-Algorithmus bereit. Liefert die Bewertung zurück und speichert den optimalen Zug in einer globalen Variable gespeicherter_zug.
minn	Stellt die MIN-Methode für den MinMax-Algorithmus bereit.
gebe_performance_aus	Gibt die Performancewerte in einer Zeile, ggf. mit Überschrift, aus.
schlage_naechsten_zug_vor	Schlägt den nächsten Zug für einen gegebenen Spieler vor.

6.5.2 Funktion zur Nutzensbewertung

Verbale Beschreibung des Algorithmus:

- *Wenn bereits eine Gewinnposition erkannt wird, gebe (+/-) 50.000 zurück.*
- *Ansonsten soll folgende Heuristik eingesetzt werden: Zähle die Anzahl der Einer, Zweier, Dreier pro Zeile. Multipliziere die Anzahl der Einer mit 1, die Anzahl der Zweier mit 15 und die Anzahl der Dreier mit 400.*
- *Summiere die Anzahl pro Spieler.*
- *Gewichte den Computerspieler 10% stärker, um ein Unentschieden zu vermeiden.*
- *Bilde die Differenz aus den Werten für die Spieler.*

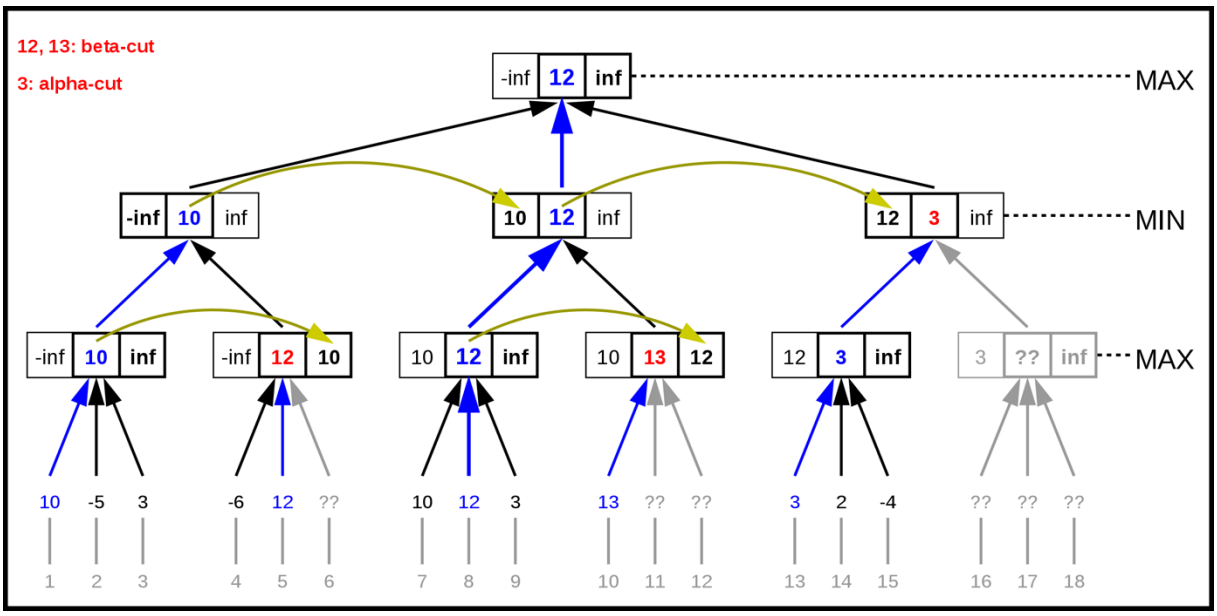
6.5.3 MinMax-Algorithmus mit Alpha-Beta-Erweiterung

Die Alpha-Beta-Suche ignoriert alle Knoten, für die beim Erreichen klar ist, dass sie das Suchergebnis nicht beeinflussen können.[25]

[25] Das Bild und die zugehörige Beschreibung wurden aus (wikipedia - Alpha-Beta-Suche, 2022) entnommen.

Der Spielbaum in der Abbildung hat 18 Blätter, von denen nur 12 ausgewertet werden. Die drei umrandeten Werte eines inneren Knotens stehen für den Alpha-Wert, den Rückgabewert und den Beta-Wert. Der Suchalgorithmus verwendet ein sogenanntes Alpha-Beta-Fenster, mit dem Alpha-Wert als untere und dem Beta-Wert als obere Grenze. Dieses Fenster wird zu den Folgeknoten weitergegeben, wobei in der Wurzel mit dem maximalen Fenster [-unendlich, unendlich] begonnen wird. Die Blätter 1, 2 und 3 werden von einem Max-Knoten ausgewertet und der beste Wert 10 wird dem Min-Vaterknoten übergeben. Dieser passt den Beta-Wert an und übergibt das neue Fenster [-unendlich, 10] dem nächsten Max-Folgeknoten, der die Blätter 4, 5 und 6 besitzt. Der Rückgabewert 12 von Blatt 5 ist aber so gut, dass er den Beta-Wert 10 überschreitet. Somit muss Blatt 6 nicht mehr betrachtet werden, weil das Ergebnis 12 dieses Teilbaumes besser ist als das des linken Teilbaumes und deshalb vom Min-Spieler nie gewählt werden würde.[26]

Abbildung 6.5: Alpha-Beta-Suche[27]



Ähnlich verhält es sich beim Min-Knoten mit dem 3-Alpha-Cutoff. Obwohl dieser Teilbaum erst teilweise ausgewertet wurde, ist klar, dass der Max-Wurzelknoten diese Variante niemals wählen würde, weil der Min-Knoten ein Ergebnis von höchstens 3 erzwingen könnte, während aus dem mittleren Teilbaum das Ergebnis 12 sichergestellt ist.[28]

---

[26] Siehe: (wikipedia - Alpha-Beta-Suche, 2022).
[27] Bildquelle: (Antonsusi, 2004)
[28] Siehe: (wikipedia - Alpha-Beta-Suche, 2022).

Abbildung 6.6: MinMax-Algorithmus mit Alpha-Beta-Erweiterung (Hauptprogramm)

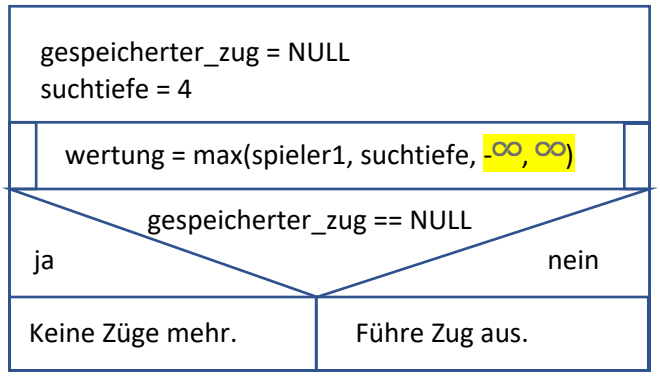


Abbildung 6.7: Max-Funktion des MinMax-Algorithmus mit Alpha-Beta-Erweiterung

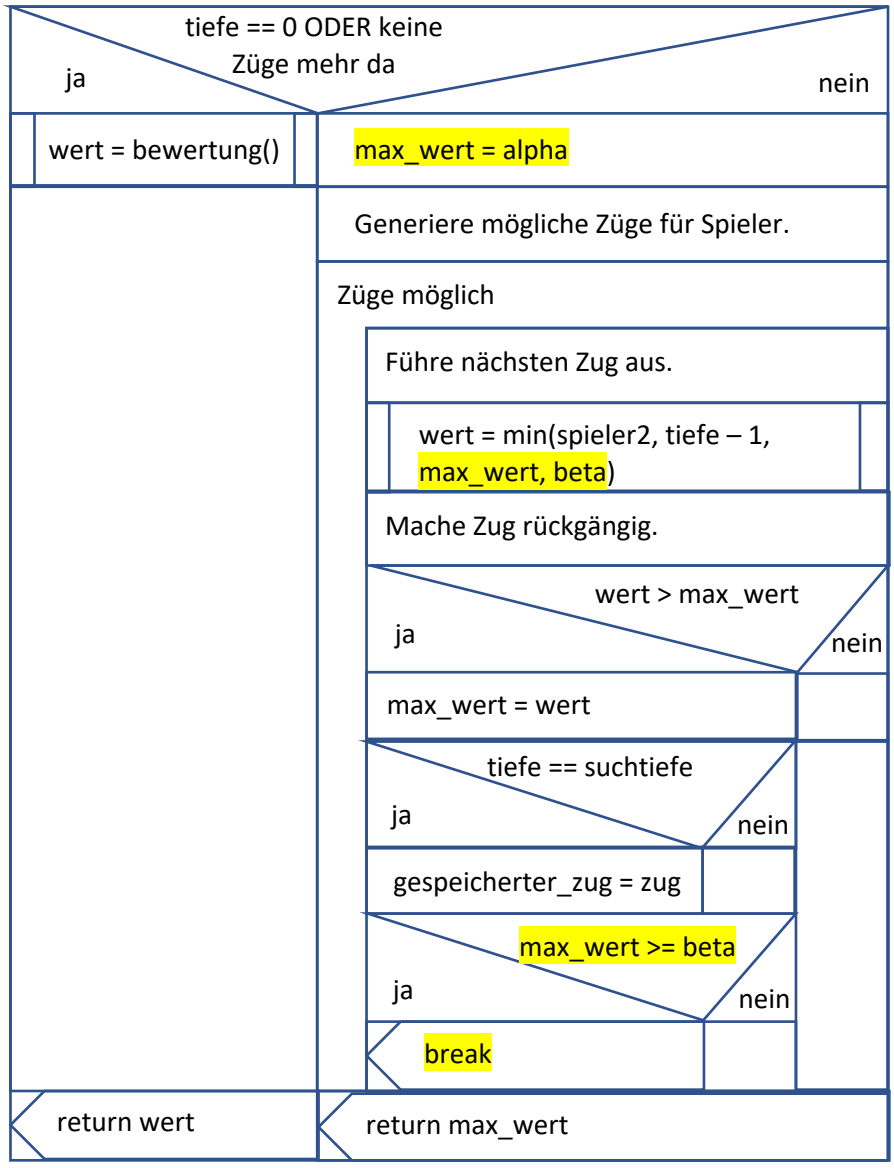
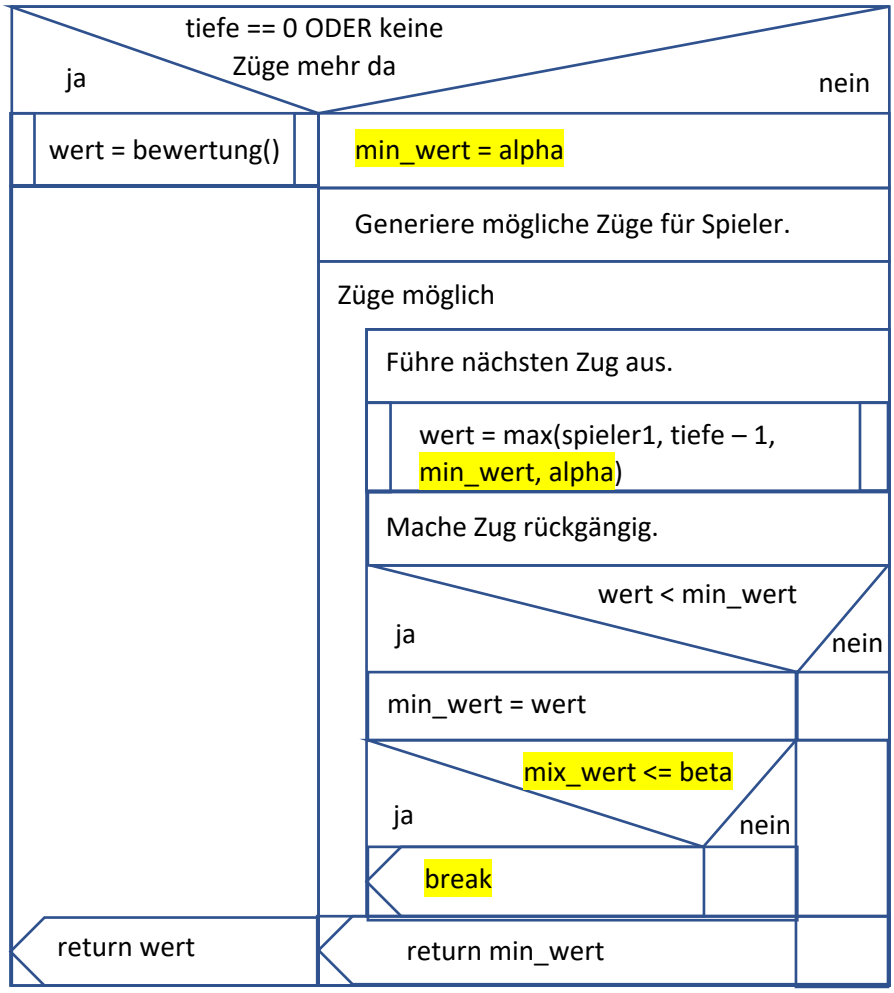


Abbildung 6.8: Min-Funktion des MinMax-Algorithmus mit Alpha-Beta-Erweiterung



6.5.4 Programmcode

```
#! /usr/bin/env python

"""
Funktionen für eine künstliche Intelligenz für ein "4 gewinnt"-Spiel
"""
__author__ = "Henriette Schulz"
__version__ = "1.0.0"
__email__ = "schulz.henriette.0106@gmail.com"
__status__ = "Prototyp"

import spielfeld as sf
import konstanten as ko
import spiel
import performance
import random
import numpy
import time

spieler1 = spiel.spieler1
spieler2 = spiel.spieler2

gespeicherter_zug = {"spieler": 100,
                     "zeile": 0,
                     "spalte": 0}
```

```
def bewerte_zelle(spielfeld, spieler, zelle):
    """
    Bewertet Zug für den MinMax-Algorithmus.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: ID für Spieler
    :param zelle: Zelle als Dictionary {"spieler", "zeile", "spalte"}
    :return: 10, wenn Spieler gewinnt; -10, wenn anderer Spieler gewinnt;
sonst 0
    """
    wert = 10
    if spieler != zelle["spieler"]:
        wert = -10

    if sf.ist_zug_gewinn(spielfeld, zelle["spieler"], zelle["spalte"],
zelle["zeile"]):
        # Gewinn erkannt
        return wert
    else:
        return 0


def suche_gewinnposition(spielfeld, spieler, p):
    """
    Bewertet das gesamte Spielfeld, ob ein Spieler gewonnen hat.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: ID für zu untersuchenden Spieler
    :param p: Objekt für Performanceinformationen
    :return: 10, wenn Spieler gewinnt; -10, wenn anderer Spieler gewinnt;
sonst 0
    """
    felder = sf.liefere_benutzte_felder(spielfeld)
    p.untersuchte_stellungen += 1

    for feld in felder:
        w = bewerte_zelle(spielfeld, spieler, feld)
        if w != 0:
            return w

    return 0


def zaehle_anzahl_in_zeile(spielfeld, spieler, anzahl):
    """
    Zählt die Zahl der Steine in einer Zeile für Spieler - jeweils für 1 bis
anzahl.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: ID für Spieler
    :param anzahl: Max. Anzahl der Steine, die gezählt werden sollen
    :return: Array (0...anzahl-1) mit jeweiliger Zahl der gezählten Steine
    """
    gesamt = numpy.zeros(anzahl)
    zaehler = numpy.zeros(anzahl)

    for z in range(ko.ZEILEN):
        for s in range(ko.SPALTEN):
            if spielfeld[z, s] == spieler:
                # Das Feld ist von diesem Spieler besetzt.
                # Zähle das Feld mit.

                for i in range(anzahl):
```

```
                    zaehler[i] += 1
                    if zaehler[i] == i + 1:
                        # Anzahl der Gewinnfelder ist erreicht
                        gesamt[i] += 1
            else:
                # Das nächste Feld ist von anderem Spieler besetzt.
                # Fange wieder von vorn an zu zählen.
                zaehler = numpy.zeros(anzahl)
    return gesamt


def berechne_wert(bewertung):
    """
    Berechnet einen gewichteten Wert je Anzahl der Spielsteine in einer
Zeile.
    :param bewertung: Liste mit Anzahlen für Einer, Zweier usw.
    :return: gewichteter Wert
    """
    wert = 0
    for i in range(ko.GEWINNFELDER - 1):
        wert += bewertung[i] * ko.GEWICHT[i]
    return wert


def bewerte_spiel(spielfeld, spieler, p):
    """
    Bewertet das gesamte Spielfeld, ob ein Spieler gewonnen hat.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: ID für zu untersuchenden Spieler
    :param p: Objekt für Performanceinformationen
    :return: 50000, wenn Spieler gewinnt; -50000, wenn anderer Spieler ge-
winnt; sonst Zwischenwert
    """
    wert_vier = suche_gewinnposition(spielfeld, spieler, p)
    if wert_vier != 0:
        return wert_vier * ko.GEWICHT[ko.GEWINNFELDER - 1]

    bewertung_z1 = zaehle_anzahl_in_zeile(spielfeld, spieler, ko.GEWINNFEL-
DER)
    wert_s1 = berechne_wert(bewertung_z1)
    bewertung_z2 = zaehle_anzahl_in_zeile(spielfeld, spiel.liefere_ande-
ren_spieler(spieler), ko.GEWINNFELDER)
    wert_s2 = berechne_wert(bewertung_z2)

    if spiel.liefere_name(spieler) == "Computer":
        wert_s1 = int(wert_s1 * 1.1)
    else:
        wert_s2 = int(wert_s2 * 1.1)

    wert = wert_s2 - wert_s1

    p.untersuchte_stellungen += 1

    return wert


def fuehre_naechsten_zug_aus(spielfeld, spieler, zuege, index):
    """
    Führt den nächsten Zug aus einer Liste von Zügen aus.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: ID für Spieler
    :param zuege: Liste der möglichen Züge
```

```
    :param index: Index für den nächsten Zug
    :return: nächster Zug als Dictionary {"spieler", "zeile", "spalte"}
    """
    if bool(zuege):
        naechster_zug = zuege.pop(index)
        spalte = naechster_zug["spalte"]
        sf.mache_zug(spielfeld, spieler, spalte)

        return naechster_zug
    else:
        return None


def maxx(spielfeld, spieler, tiefe, alpha, beta, p):
    """
    Stellt die MAX-Methode für den MinMax-Algorithmus bereit.
    Liefert die Bewertung zurück und speichert den optimalen Zug in einer
globalen Variable gespeicherter_zug.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: ID für Spieler
    :param tiefe: Suchtiefe
    :param alpha: Parameter für Alpha-Beta-Suche
    :param beta: Parameter für Alpha-Beta-Suche
    :param p: Objekt für Performanceinformationen
    :return: Bewertung des Spielstandes
    """
    moegliche_zuege = spiel.erzeuge_moegliche_zuege(spielfeld)

    p.max_aufrufe += 1

    if (tiefe == 0) or (not bool(moegliche_zuege)):
        wert = bewerte_spiel(spielfeld, spieler, p)
        return wert

    max_wert = alpha

    while bool(moegliche_zuege):
        index = int(len(moegliche_zuege) / 2)
        zug = moegliche_zuege[index]
        fuehre_naechsten_zug_aus(spielfeld, spieler, moegliche_zuege, index)
        wert = minn(spielfeld, spiel.liefere_anderen_spieler(spieler), tiefe
- 1, max_wert, beta, p)
        sf.nehme_zug_zurueck(spielfeld, zug)

        if wert > max_wert:
            max_wert = wert
            if tiefe == ko.TIEFE:
                global gespeicherter_zug
                gespeicherter_zug = zug
            if max_wert >= beta:
                break

    return max_wert


def minn(spielfeld, spieler, tiefe, alpha, beta, p):
    """
    Stellt die MIN-Methode für den MinMax-Algorithmus bereit.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: ID für Spieler
    :param tiefe: Suchtiefe
    :param alpha: Parameter für Alpha-Beta-Suche
```

```
    :param beta: Parameter für Alpha-Beta-Suche
    :param p: Objekt für Performanceinformationen
    :return: Bewertung des Spielstandes
    """
    p.min_aufrufe += 1
    moegliche_zuege = spiel.erzeuge_moegliche_zuege(spielfeld)

    if (tiefe == 0) or (not bool(moegliche_zuege)):
        wert = bewerte_spiel(spielfeld, spieler, p)
        return wert

    min_wert = beta

    while bool(moegliche_zuege):
        index = int(len(moegliche_zuege) / 2)
        zug = moegliche_zuege[index]
        fuehre_naechsten_zug_aus(spielfeld, spieler, moegliche_zuege, index)
        wert = maxx(spielfeld, spiel.liefere_anderen_spieler(spieler), tiefe
- 1, alpha, min_wert, p)
        sf.nehme_zug_zurueck(spielfeld, zug)

        if wert < min_wert:
            min_wert = wert
            if min_wert <= alpha:
                break

    return min_wert


def gebe_performance_aus(zug_spalte, max_wert, p, laufzeit, spieler, zufall,
erster_zug=False):
    """
    Gibt die Performancewerte in einer Zeile, ggf. mit Überschrift, aus.
    :param zug_spalte: Spalte des Zuges
    :param max_wert: Wert der MinMax-Funktion
    :param p: Objekt mit Performancewerten
    :param laufzeit: Laufzeit
    :param spieler: ID für Spieler
    :param zufall: True, wenn zufälliger Zug, sonst False
    :param erster_zug: Wenn True, wird Überschrift ausgegeben.
    :return: nichts
    """
    # bereite Überschrift vor
    if erster_zug:
        ueberschriften = ["Zug in Spalte", "Gewinnzug", "Zufall", "Max.
Wert", "Aufrufe Max", "Aufrufe Min",
                          "Stellungen", "Laufzeit [s]"]
        ueberschrift = ""
        for u in ueberschriften:
            ueberschrift += f"{u:<15}" + "\t"

        print(ko.FETT + ueberschrift + ko.FETT_ENDE)

    # bereite Ergebniszeile vor
    laufzeit_string = f"{laufzeit:.2f}"
    zufall_string = "n/a"
    if zufall:
        zufall_string = "ja"

    if (spieler == ko.SPIELER1) or (spieler == ko.SPIELER2):
        ergebnisse = [zug_spalte, spiel.liefere_name(spieler), zufall_string,
"n/a", 0, 0, 0, laufzeit_string]
```

```
    else:
        ergebnisse = [zug_spalte, "n/a", zufall_string, max_wert, p.max_auf-
rufe, p.min_aufrufe,
                      p.untersuchte_stellungen, laufzeit_string]

    ergebnis = ""
    for e in ergebnisse:
        ergebnis += f"{e:<15}" + "\t"

    print(ergebnis.replace('.', ','))


def schlage_naechsten_zug_vor(spielfeld, spieler, erster_zug):
    """
    Schlägt den nächsten Zug für einen gegebenen Spieler vor.
    :param spielfeld: Array mit Spielinformationen
    :param spieler: ID für Spieler
    :param erster_zug: True, wenn erster Zug, sonst False
    :return: Zug als Dictionary {"spieler", "zeile", "spalte"}
    """
    t0 = time.time()
    p = performance.PerformanceZahlen()

    moegliche_zuege = spiel.erzeuge_moegliche_zuege(spielfeld)

    # Gewinnt Spieler sofort mit diesem Zug?
    for zug in moegliche_zuege:
        zug["spieler"] = spieler
        if sf.ist_zug_gewinn(spielfeld, zug["spieler"], zug["spalte"],
zug["zeile"]):
            t1 = time.time()
            laufzeit = t1 - t0
            gebe_performance_aus(zug["spalte"], 0, p, laufzeit, spieler,
False, erster_zug)
            return zug

    # Verhindert der Zug den Gewinn des Gegners?
    for zug in moegliche_zuege:
        gegner = spiel.liefere_anderen_spieler(spieler)
        zug["spieler"] = gegner
        raus = sf.ist_zug_gewinn(spielfeld, zug["spieler"], zug["spalte"],
zug["zeile"])
        if raus:
            t1 = time.time()
            laufzeit = t1 - t0
            gebe_performance_aus(zug["spalte"], 0, p, laufzeit, gegner,
False, erster_zug)
            return zug

    max_wert = maxx(spielfeld, spieler, ko.TIEFE, -65000, 65000, p)

    t1 = time.time()
    laufzeit = t1 - t0

    if int(max_wert) == 0:
        # keine Gewinnstellung erkannt, mache zufälligen Zug
        zufall = random.randint(0, len(moegliche_zuege) - 1)
        zug = moegliche_zuege[zufall]
        gebe_performance_aus(zug["spalte"], max_wert, p, laufzeit, 0, True,
erster_zug)
        return zug
```

```
    elif int(max_wert) == -50000:
    # mit dem nächsten Zug gewinnt der Gegner, also vermeiden, wenn möglich
    verlust = gespeicherter_zug["spalte"]
    # Falls nur noch ein Zug möglich ist, dann muss er gemacht werden. Sonst
vermeiden.
    if len(moegliche_zuege) <= 1:
        zug = gespeicherter_zug
    else:
        # Suche den Verlustzug aus den möglichen Zügen heraus und eliminiere
ihn.
        for i in range(len(moegliche_zuege)):
            mz = moegliche_zuege[i]
            if mz["spalte"] == verlust:
                moegliche_zuege.pop(i)
        # Wenn nur noch ein Zug übrig bleibt, mache diesen. Sonst zufälligen
möglichen Zug machen.
        if len(moegliche_zuege) <= 1:
            index = 0
        else:
            index = random.randint(0, len(moegliche_zuege) - 1)
        zug = moegliche_zuege[index]
    gebe_performance_aus(zug["spalte"], max_wert, p, laufzeit, 0, True, ers-
ter_zug)
    return zug


    else:
        gebe_performance_aus(gespeicherter_zug["spalte"], max_wert, p, lauf-
zeit, 0, False, erster_zug)

    return gespeicherter_zug
```

6.6 Globale Konstanten

```
#! /usr/bin/env python

"""
Definition von globalen Konstanten für ein "4 gewinnt"-Spiel
"""
__author__ = "Henriette Schulz"
__version__ = "1.0.0"
__email__ = "schulz.henriette.0106@gmail.com"
__status__ = "Prototyp"

TIEFE = 6  # Suchtiefe

SPALTEN = 7  # Spaltenzahl
ZEILEN = 6  # Zeilenzahl
GEWINNFELDER = 4  # Anzahl der Gewinnfelder

GEWICHT = [1, 15, 400, 5000]  # Gewichte für Bewertungsfunktion

RADIUS = 30  # Radius für Kreise auf dem Spielfeld
ABSTAND = 10  # Abstand zwischen den Kreisen und zum Rand

W = SPALTEN * (2 * RADIUS + ABSTAND) + ABSTAND  # Breite des Spielfelds in
Pixels
H = ZEILEN * (2 * RADIUS + ABSTAND) + ABSTAND + 100  # Höhe des Spielfeldes
in Pixels
FPS = 60  # Bildwiederholrate für Fenster
```

```
# Farbdefinitionen
WEISS = (255, 255, 255)
SCHWARZ = (0, 0, 0)
GRAU = (220, 220, 220)
ROT = (237, 125, 49)
BLAU = (68, 114, 196)

FETT = '\033[1m'  # fette Ausgabe von Strings
FETT_ENDE = '\033[0m'  # Ende der fetten Ausgabe

SPIELER1 = 1  # ID für Spieler 1
SPIELER2 = 2  # ID für Spieler 2

KI_MODUS = True  # Computer soll Züge machen

if KI_MODUS:
    SPIELERNAME2 = "Spieler"
    SPIELERNAME1 = "Computer"  # Spieler 1 beginnt immer, wenn "Computer",
dann beginnt dieser
else:
    SPIELERNAME1 = "Spieler 1"  # Name für Spieler 1
    SPIELERNAME2 = "Spieler 2"  # Name für Spieler 2
```

6.7 Klasse für Performancedaten

6.7.1 Funktionen

Da erwartet wird, dass der Algorithmus zur Ermittlung eines Zugvorschlags häufig durchlaufen wird und durch die rekursive Abarbeitung die Häufigkeit schlecht abschätzbar ist, wurden Funktionen implementiert, um sowohl die Anzahl der Aufrufe als auch die Laufzeiten des Algorithmus aufzuzeichnen.

Dazu wird bei jedem Aufruf eine Referenz auf ein entsprechendes Objekt (enthält nur die zugehörige Datenstruktur) mitgegeben, in dem dann pro Aufruf die entsprechenden Werte hochgezählt werden.[29]

Abbildung 6.9 zeigt die Ausgabe der entsprechenden Daten auf der Konsole für ein beispielhaft gegen den Computer geführtes Spiel.

[29] Da es sich hier nicht um den eigentlichen Algorithmus handelt, wurde eine Klasse mit zugehörigem Objekt implementiert, da sich hiermit die Übergabe mittels „Call by Reference“ elegant realisieren ließ.

Abbildung 6.9: "Telemetriedaten" für ein Spiel

Zug in Spalte	Gewinnzug	Zufall	Max. Wert	Aufrufe Max	Aufrufe Min	Stellungen	Laufzeit [s]
3	n/a	n/a	1.0	1375	563	2162	0.08
2	n/a	n/a	15.0	3987	1111	6673	0.24
3	n/a	n/a	15.0	2345	777	3863	0.16
3	n/a	n/a	15.0	1682	626	2895	0.13
5	Spieler	n/a	n/a	0	0	0	0.00
5	n/a	n/a	13.0	1249	488	1946	0.13
4	n/a	n/a	-1.0	1451	590	2076	0.14
2	n/a	n/a	12.0	1574	596	2353	0.15
5	Spieler	n/a	n/a	0	0	0	0.00
6	Spieler	n/a	n/a	0	0	0	0.00
2	n/a	n/a	50000	269	167	212	0.01
2	Computer	n/a	n/a	0	0	0	0.00

6.7.2 Programmcode

```
#! /usr/bin/env python

"""
Objekt zum Speichern von Performancezahlen für ein "4 gewinnt"-Spiel
"""
__author__ = "Henriette Schulz"
__version__ = "1.0.0"
__email__ = "schulz.henriette.0106@gmail.com"
__status__ = "Prototyp"


class PerformanceZahlen(object):
    def __init__(self):
        self.min_aufrufe = 0
        self.max_aufrufe = 0
        self.untersuchte_stellungen = 0
```

6.8 Erweiterungsmöglichkeiten

Tabelle 6.5: Erweiterungsmöglichkeiten

Funktion	Erklärung
Darstellung der Gewinnstellung	Die Steine, die zur Gewinnstellung führen, sollten optisch hervorgehoben werden, um so besser erkennbar zu sein.
Einstellungen durch den Nutzer	Bisher sind die Einstellungen über Konstanten im Programmcode realisiert. Wünschenswert wären Einstellungen über das Spielfenster, so dass der Nutzer sie direkt vornehmen und abspeichern kann (z.B. Auswahl der Spielstärke, Auswahl des beginnenden Spielers, Auswahl des KI-Modus).
Zug zurücknehmen	Dem Spieler sollte die Möglichkeit gegeben werden, einen versehentlich gemachten Zug zurückzunehmen.

Spielstand speichern und laden	Um das Spiel unterbrechen zu können, sollte der Spielstand gespeichert und wieder geladen werden können.

Versicherung der selbstständigen Anfertigung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle Textstellen, die wörtlich oder sinngemäß aus veröffentlichten oder nichtveröffentlichten Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

_____

Walldorf, den 4. März 2023