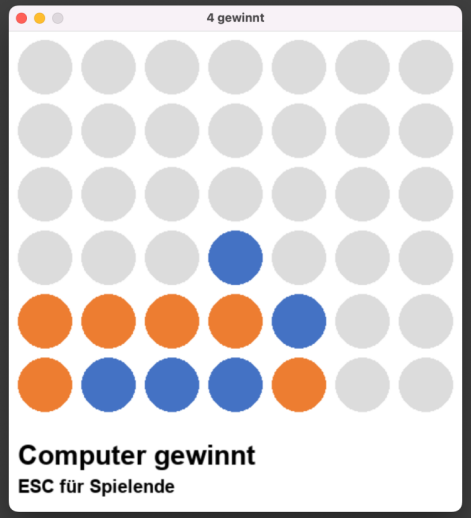
```python
def pruefe_spalte_fuer_gewinn(spielfeld, spieler, spalte, ze
    """
    Prüft in gegebener Spalte, ob Gewinnposition für Spieler
    :param spielfeld: Array mit Spielinformationen
    :param spieler: Spieler
    :param spalte: zu untersuchende Spalte
    :param zeile: Zeile, auf die nächster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



Python im Alltag

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

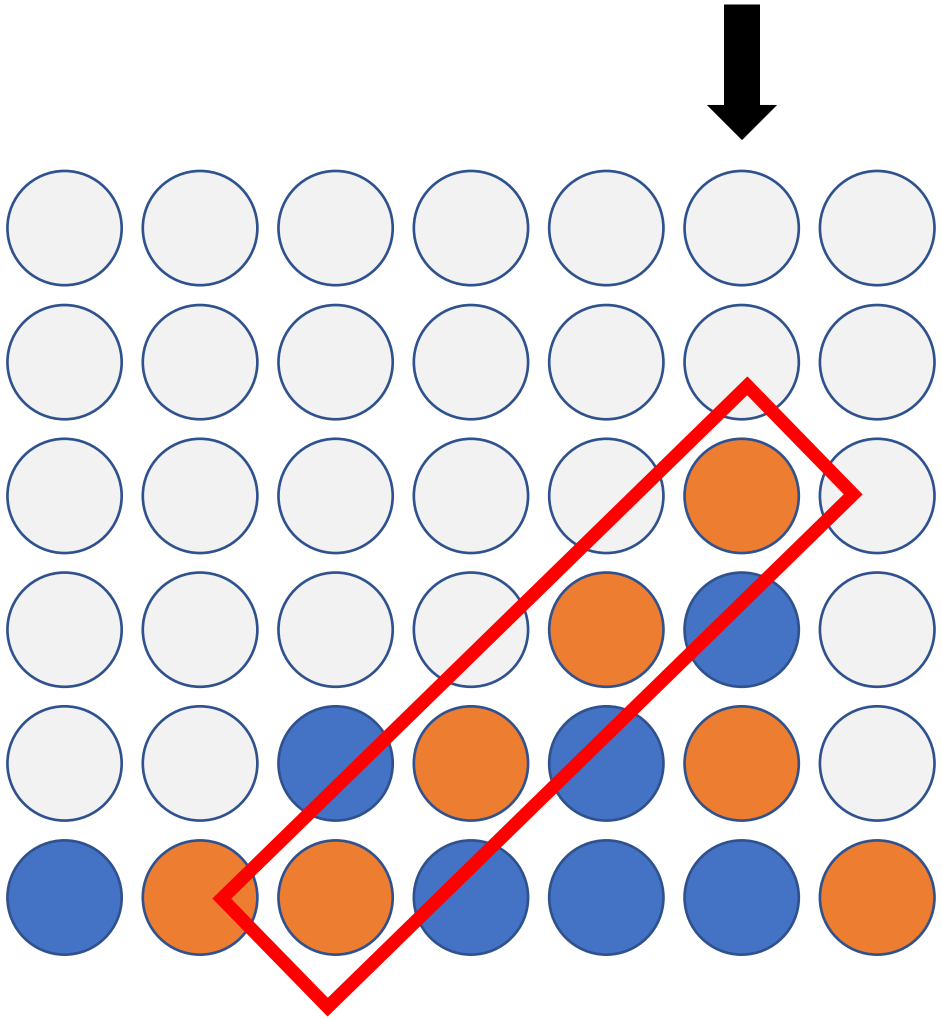
Henriette Schulz

Das Spiel

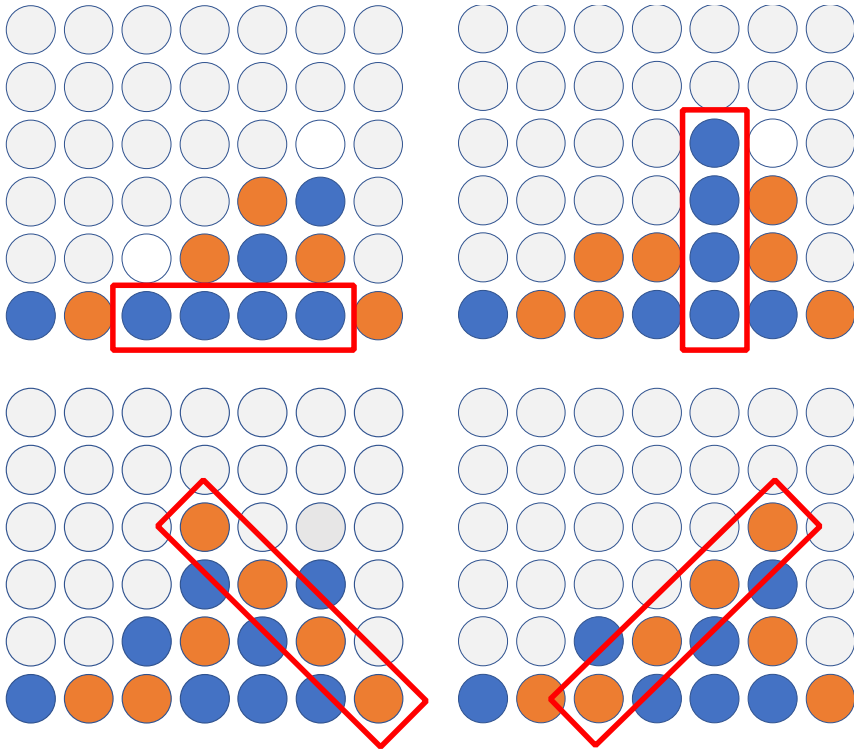
Worum geht es überhaupt?

Bildquelle: Hasbro.com

Das Spiel



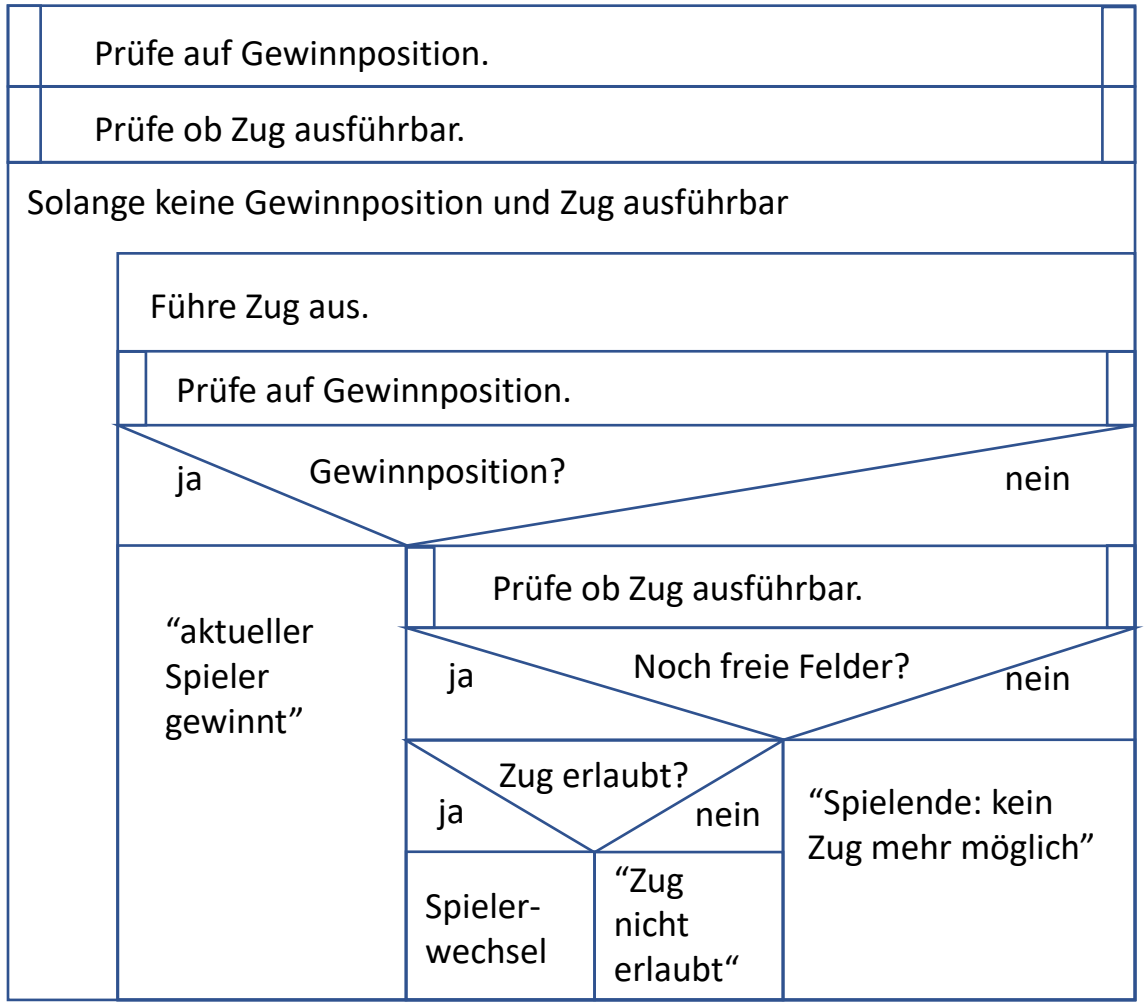
- Spielidee von Howard Wexler und Ned Strongin; heute unter Lizenz von Hasbro
- Spielfeld von 7 * 6 Feldern
- Spieler lassen abwechselnd Steine von oben hineinfallen
- Spieler gewinnt, wenn er vier Steine zusammenhängend in einer Zeile, Spalte oder Diagonale hat



Das Spiel als Algorithmus

Zwei Spieler nutzen den Computer, um gegeneinander zu spielen.

Der Spielablauf als Algorithmus



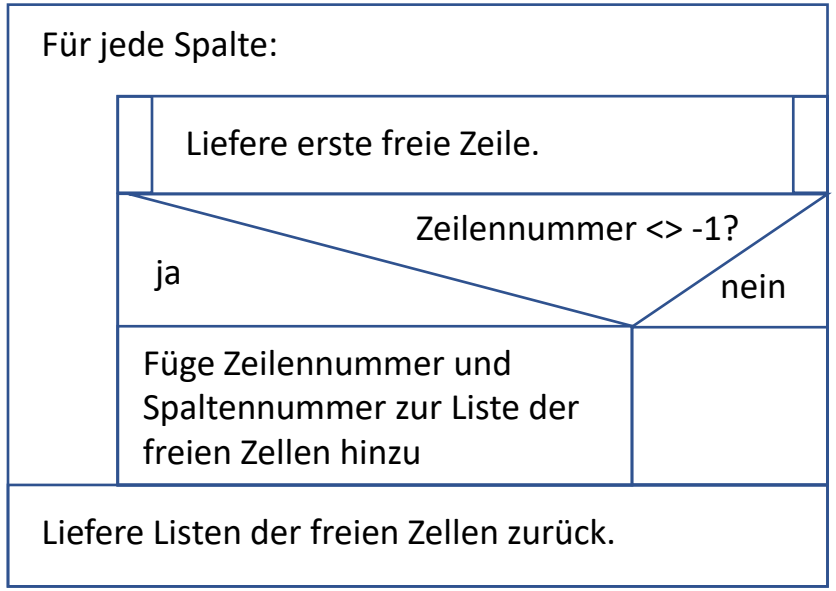
Idee: Spiele das Spiel so lange wie jemand Steine setzen kann und nicht gewonnen hat.

- Solange Spiel weiter gespielt werden kann:
  - Führe Zug aus*
  - Gewinnposition erreicht?
    - Ja: "Spielende: Aktueller Spieler gewinnt."
    - Nein: Weiterer Zug möglich?
      - Ja: Zug erlaubt? **
        - Ja: Spielerwechsel
        - Nein: „Zug nicht erlaubt"
      - Nein: "Spielende: Kein Zug mehr möglich."

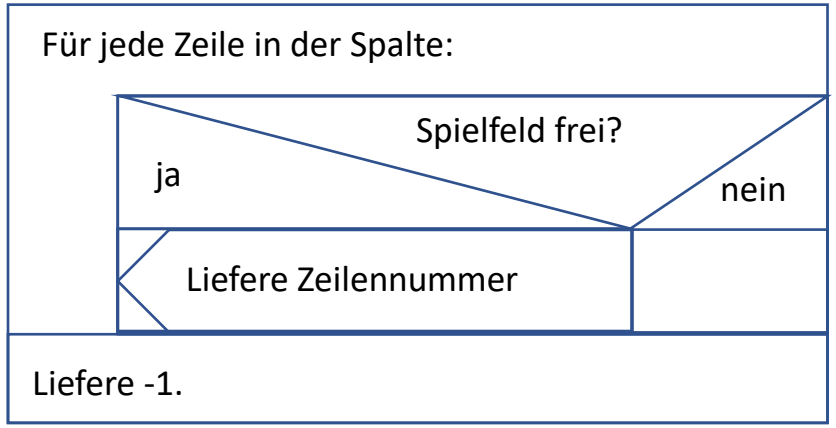
* ggf. mit Vorschlag des Zuges durch den Computer

** Versuch abfangen, Stein in volle Spalte zu setzen

Weitere Züge möglich?



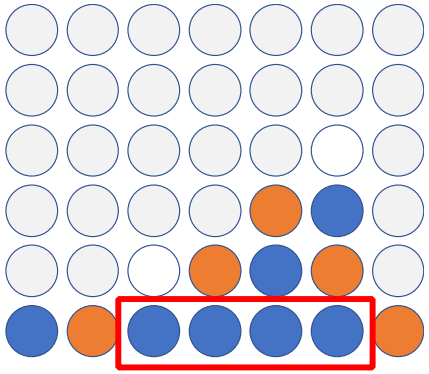
Liefere erste freie Zeile:



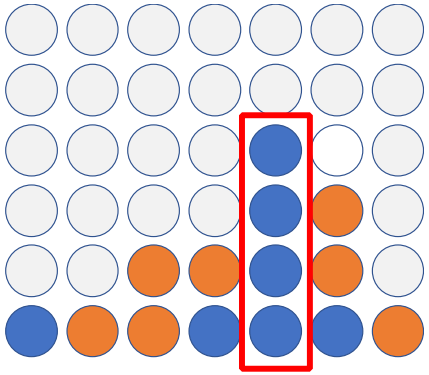
Idee: Ein Zug ist dann möglich, wenn es noch freie Felder in den Spalten gibt.

- Ermittlung der möglichen Züge:
  - Für jede Spalte:
    - Ermittle erste freie Zeile:
      - Für jede Zeile (von unten nach oben):
        - Spielfeld frei?
          - Ja: Liefere Zeilennummer, Abbruch
      - Liefere -1
    - Freie Zeile existiert?
      - Ja: Füge Zeilennummer und Spaltennummer zu Liste der freien Zellen hinzu.
  - Gebe Liste der freien Zellen zurück.

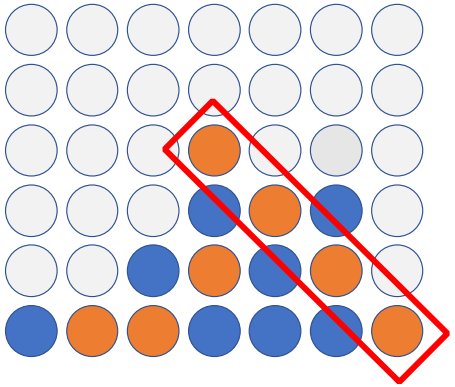
Erkennen der Gewinnposition



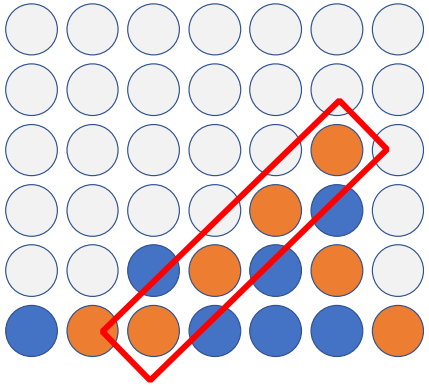
4 in einer Zeile



4 in einer Spalte



4 in einer Diagonale von links oben nach rechts unten

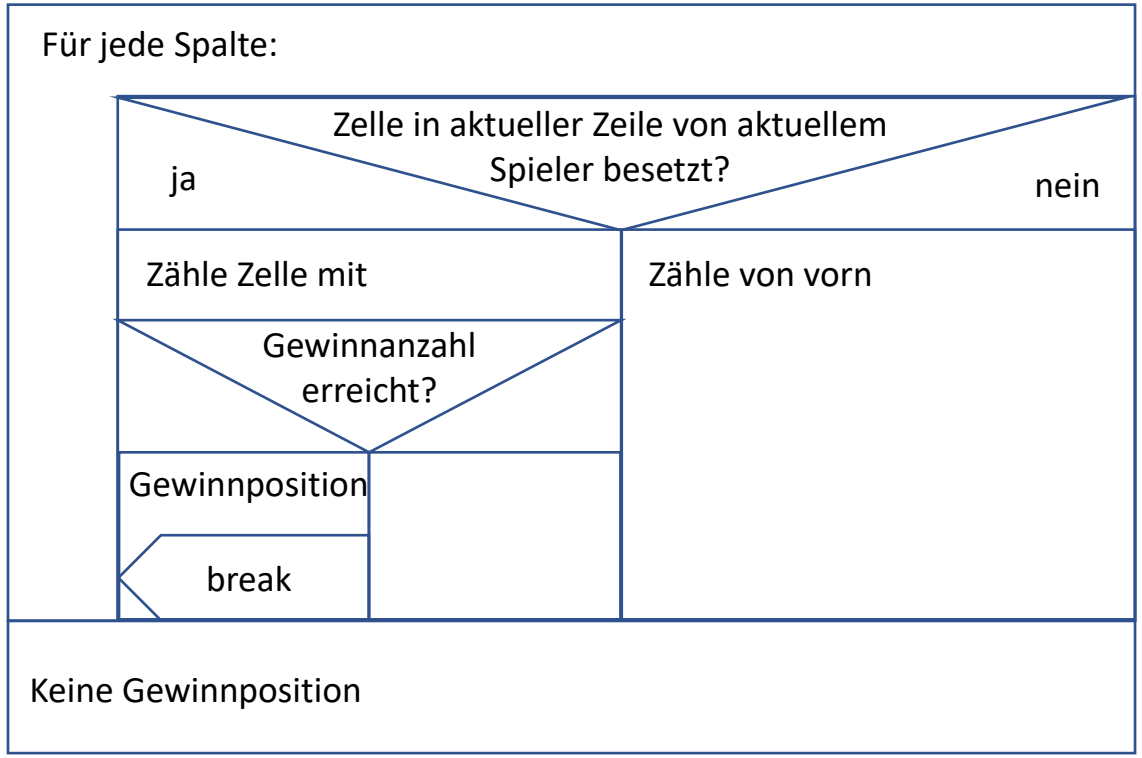


4 in einer Diagonale von rechts oben nach links unten

Ideen:

- ***Es gibt vier grundsätzliche Gewinnstellungen (siehe links).***
- ***Es wird immer für den aktuellen Spieler geprüft, nachdem er den Zug gemacht hat.***
- ***Die Gewinnstellung hat einen Bezug zum letzten Zug und hilft, die zu prüfenden Felder einzugrenzen.***

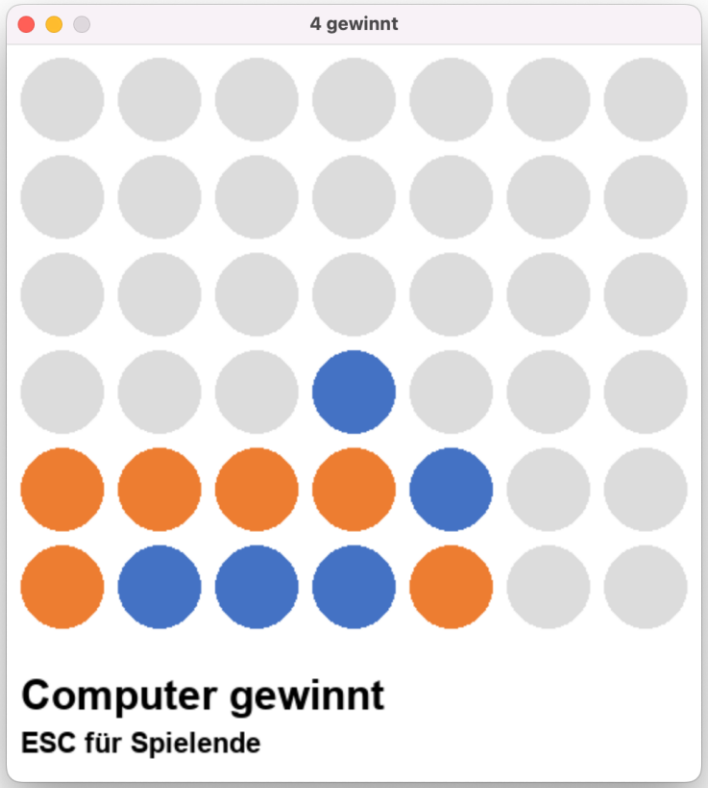
Beispiel: Gewinnposition in Zeile?



Idee:

- ***Prüfe Gewinnposition für Zeile, wo der aktuelle Stein hinfällt.***
- ***Prüfe für jedes Feld innerhalb der Zeile, ob es vom aktuellen Spieler besetzt ist.***
- Ermittlung der Gewinnposition:
  - Für jede Spalte:
    - Ist Zelle der Zeile von Spieler besetzt?
      - Ja:
        - Zähle Zelle mit.
        - Gewinnanzahl erreicht?
          - Ja: Gewinnposition, Abbruch.
      - Nein: Zähle von vorn.
  - Keine Gewinnposition

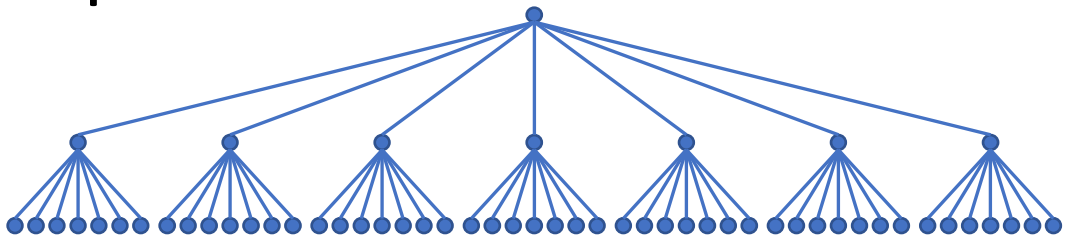
Die Ermittlung in Spalten und Diagonalen erfolgt entsprechend.



Erweiterung: “künstliche Intelligenz”

Der Computer spielt mit.

Spieltheorie



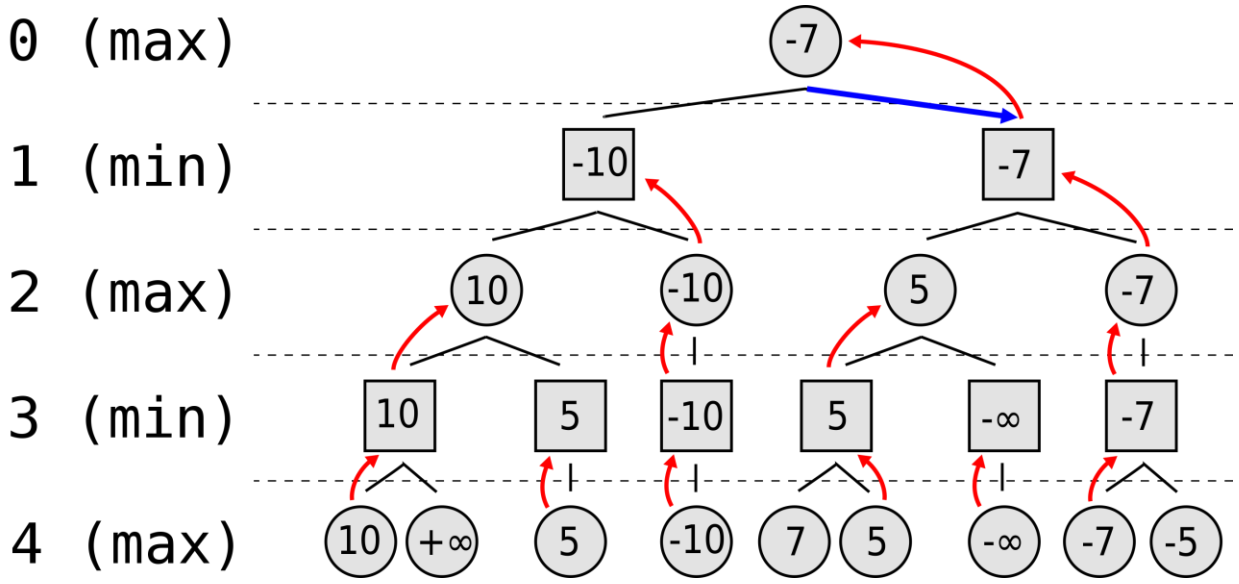
ABER:

- Nach zwei Zügen gibt es bereits 7 * 7 = 49 Möglichkeiten.
- Es gibt ca. 4,5 Billionen Spielstellungen. Woher kommt die Zahl?
  - Maximal 21 Runden, damit max. 7^{21} Möglichkeiten bzw.
  - Ein Feld kann 3 Zustände einnehmen (Spieler 1, Spieler 2, leer), damit max. 3^{42} unterschiedliche Spielstellungen
  - Aber jeweils nicht alle Möglichkeiten mit realen Zügen darstellbar.

- **Komplette Untersuchung dauert zu lange.**
- **Eine Vorberechnung und Speicherung aller Spielstände kostet zu viel Speicherplatz.**

- **Nullsummenspiel**: Die Summe der Gewinne und Verluste aller Spieler ist Null.
- Es liegt die vollständige Information vor.
- Das Spiel ist frei von Zufällen.
- Es muss demnach eine **berechenbare Gewinnstrategie** geben.

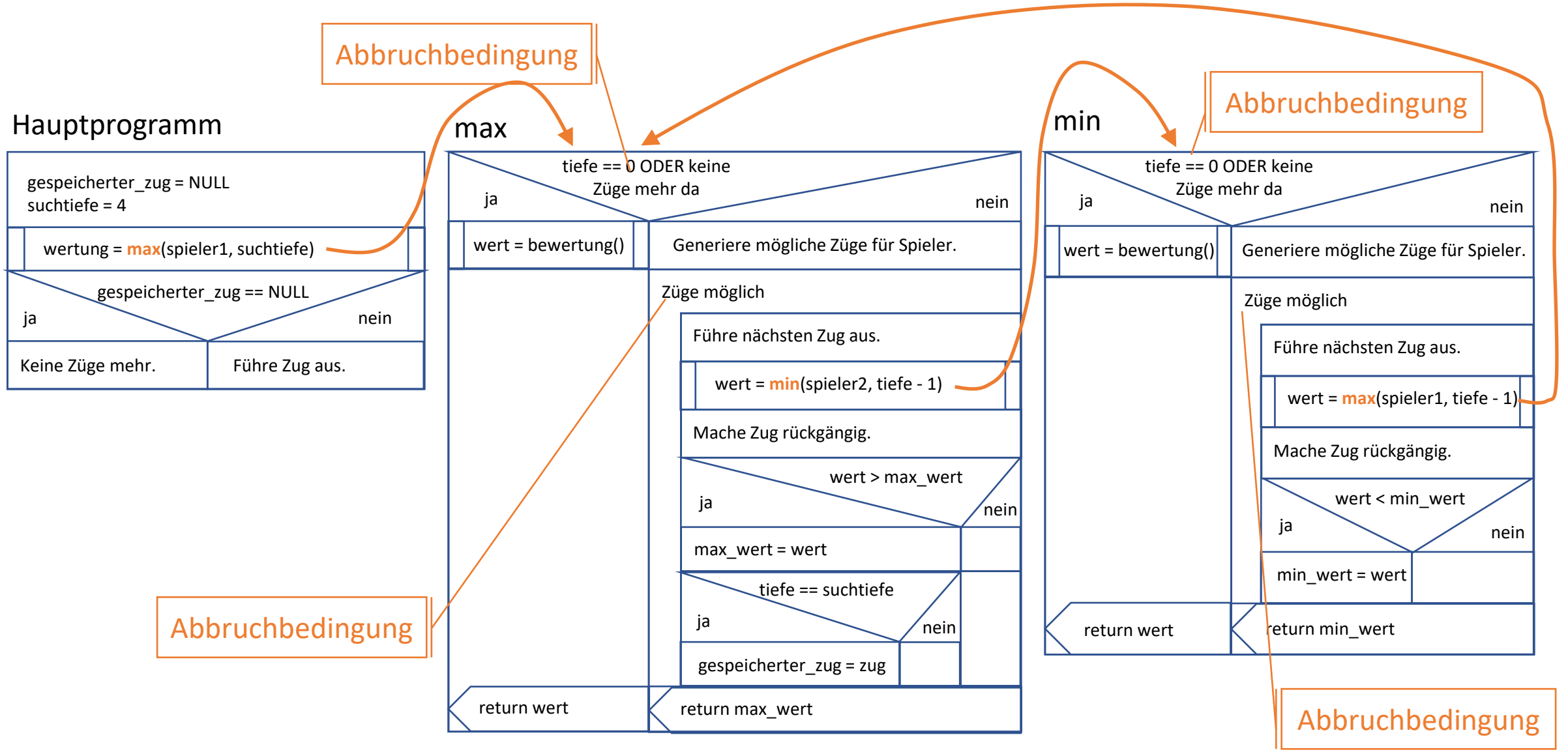
Der MinMax-Algorithmus



Bildquelle: [wikipedia.de](), Autor: Nuno Nogueira

- Zwei Spieler (**Max** und sein Gegner **Min**) **ziehen abwechselnd**.
- Jedes Mal wird die **Spielposition bewertet**.
- Der Zug mit der **besten Bewertung für den jeweiligen Spieler** soll genommen werden.
- Die **Suchtiefe** wird **begrenzt**.
- Die Suche **beginnt bei den unteren Blättern** und geht zur Wurzel.
- Der Algorithmus ist damit eine **Tiefensuche** in einem Baum.

Der Algorithmus



Bewertungsfunktion

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

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

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

Spielstrategien als Abkürzungen

- In bestimmten Situationen muss nicht der ganze Suchbaum untersucht werden:
  - Wenn der **nächste Zug einen Gewinn** ermöglicht, **mache den Zug**.
  - Wenn der **nächste Zug des Gegners einen Gewinn für ihn** ermöglicht, **verhindere den Zug**.
- Es ist von **Vorteil, Steine in der Mitte** zu setzen. Am Rand haben sie weniger Einfluss. Darum sollen zunächst die Züge in der Mitte untersucht werden.
- Wenn **kein Vorteil erkannt** wurde, mache einen **zufälligen Zug**.

```
def pruefe_spalte_fuer_gewinn(spielfeld, spieler, spalte, ze
    """
    Prüft in gegebener Spalte, ob Gewinnposition für Spieler
    :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 == ze
            # Das Feld ist von diesem Spieler besetzt.
            # Zähle Feld mit.
            gesamt += 1
            if gesamt == ko.GEWINNFELDER:
                # Anzahl der Gewinnfelder ist err
                return True
        else:
            # Das nächste Feld ist von and
            # Fange wieder von vorn an
            gesamt = 0
    return False
```

Umsetzung in Python

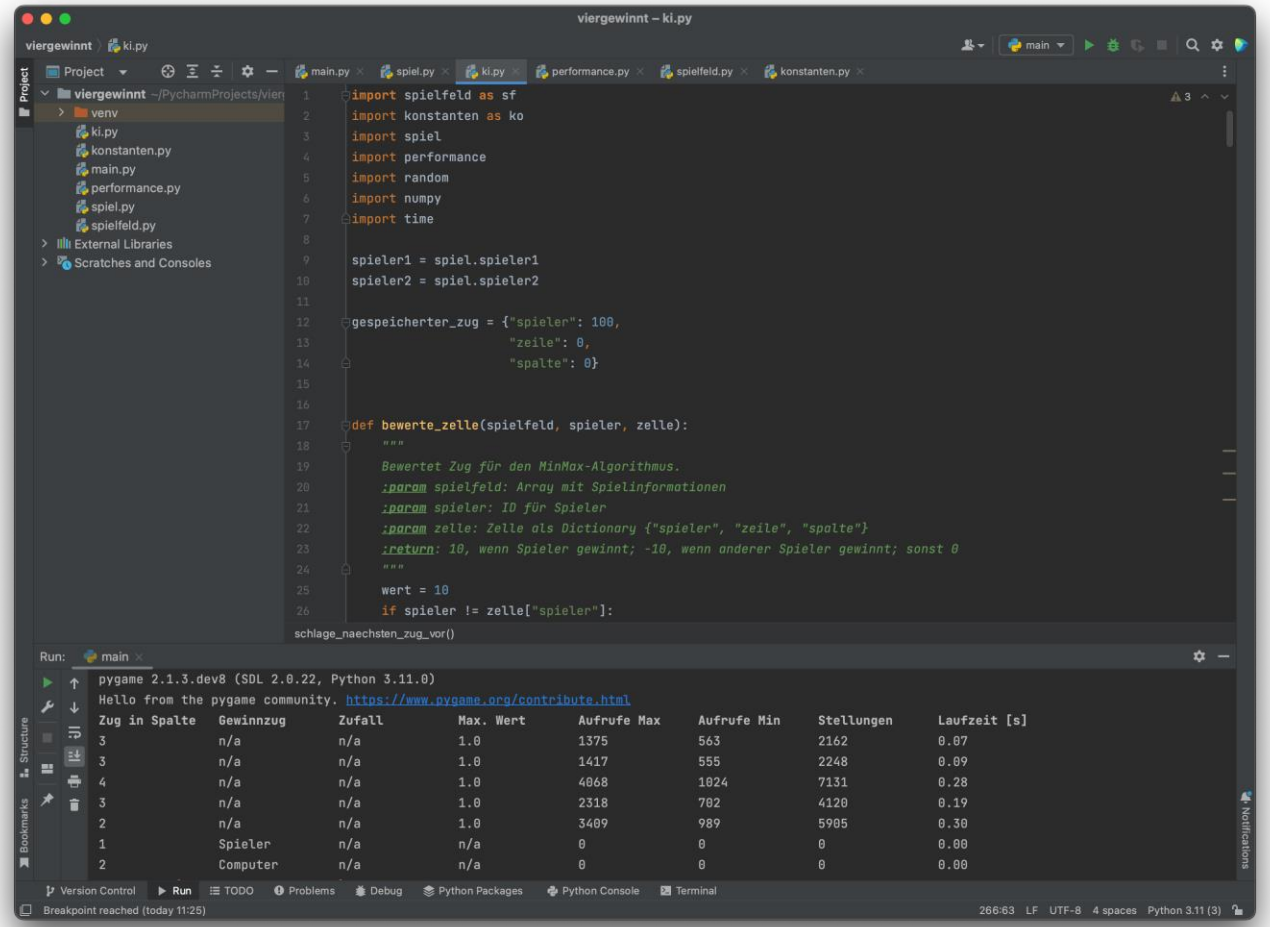
Vom Algorithmus zum Programm.

Implementierung in Python

Rahmenbedingungen

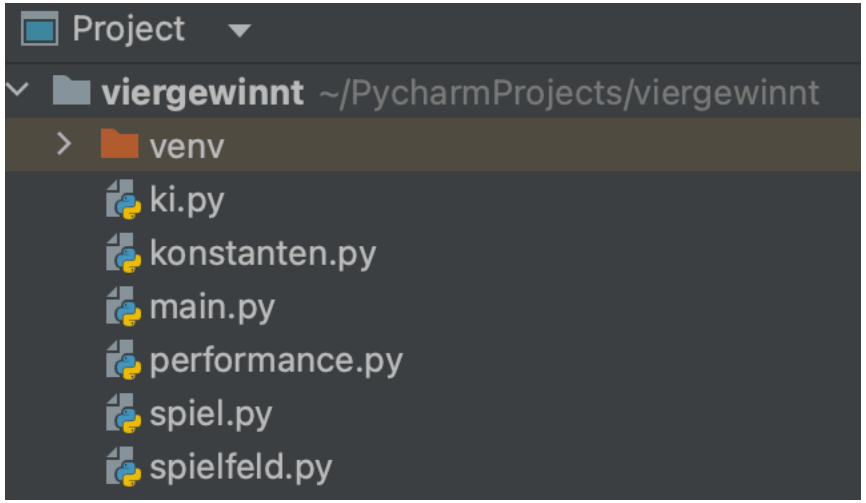
- Das Spiel möchte ich mit einer **grafischen Oberfläche** darstellen. Die Bedienung soll dabei mit der Maus erfolgen können.
- Die Implementierung soll rein unter Nutzung **prozeduraler Programmierung** erfolgen. Kein Einsatz von Objektorientierung. (Vorgabe durch die Aufgabenstellung.)
- Eigene Funktions- und Variablennamen werden, ebenso wie die Kommentare, auf **Deutsch** ausgeführt, um das Verständnis und die Lesbarkeit für Anfänger (wie uns) zu erleichtern.

Die Entwicklungsumgebung



- Benutzung von PyCharm
- Vorteile, u.a.
  - Hervorhebung der Syntax
  - leistungsfähige Editoren
  - Debugger
  - Prüfung auf korrekte Formatierung

Die Projektstruktur

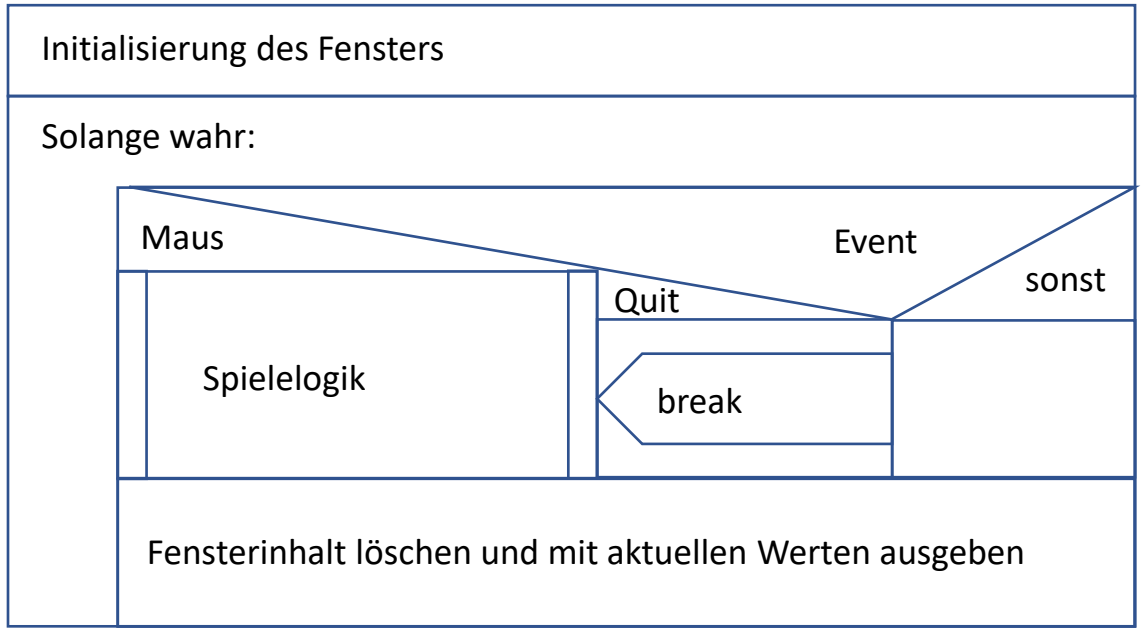


- **`main.py`**
  - Enthält die “main loop” für das Spiel und die grundlegende Spiellogik.
- **`ki.py`**
  - Enthält die Funktionen für den Computergegner.
- **`konstanten.py`**
  - Enthält die im Projekt verwendeten Konstanten.
- **`spiel.py`**
  - Enthält Logik zu Spielern (und Logik zum Berechnen 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.

pygame

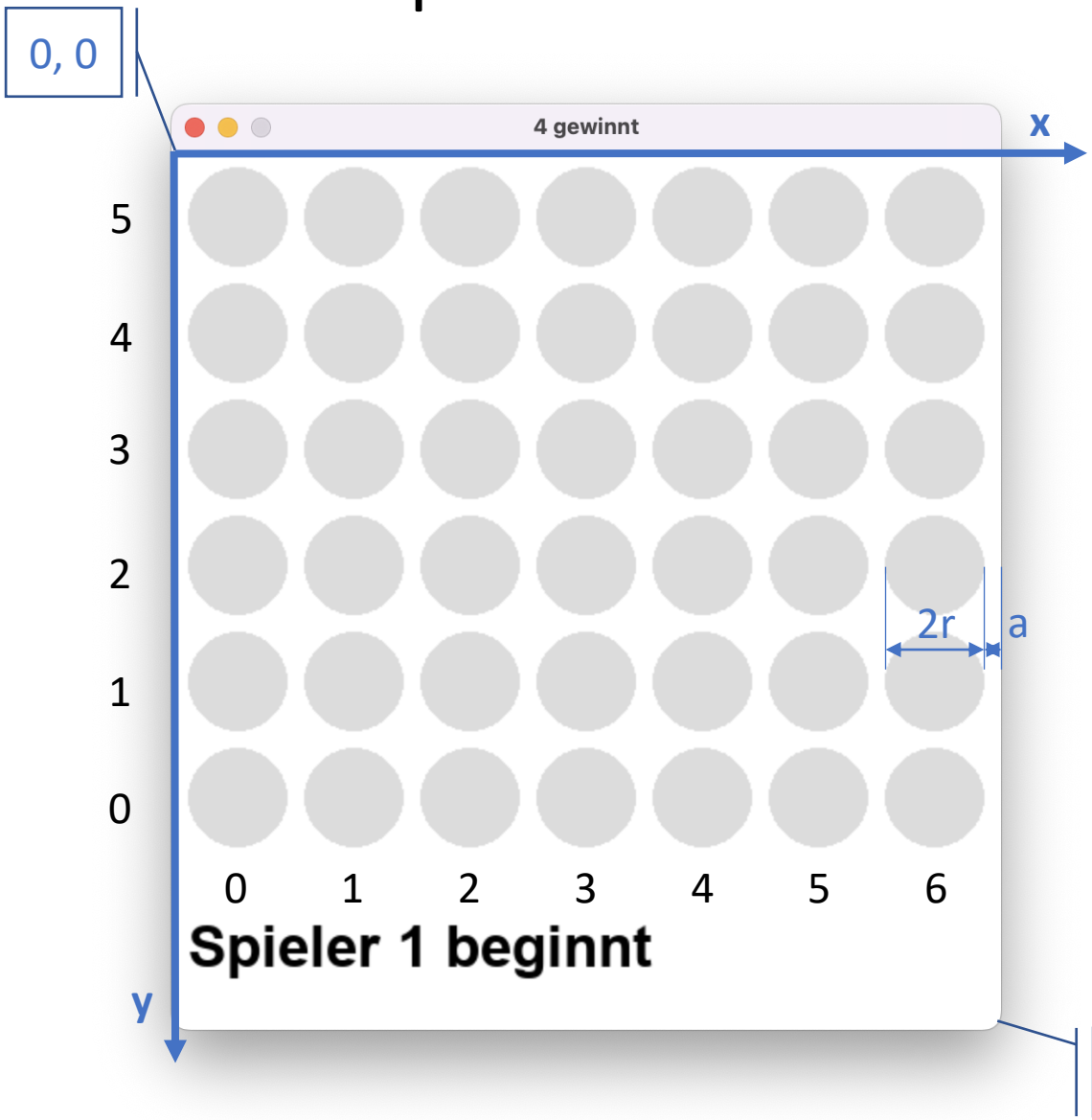
Grundsätzlicher Ablauf
(wird in bei mir `main.py` abgebildet):



- Pygame ist eine von Pete Shinners entwickelte Python-Programmbibliothek zur Spieleprogrammierung.
- Sie enthält Module zum Abspielen und Steuern von Grafik und Sound sowie zum Abfragen von Eingabegeräten (Tastatur, Maus, Joystick).

https://www.pygame.org/news
https://www.python-lernen.de/pygame-tutorial.htm

Das Spielfenster

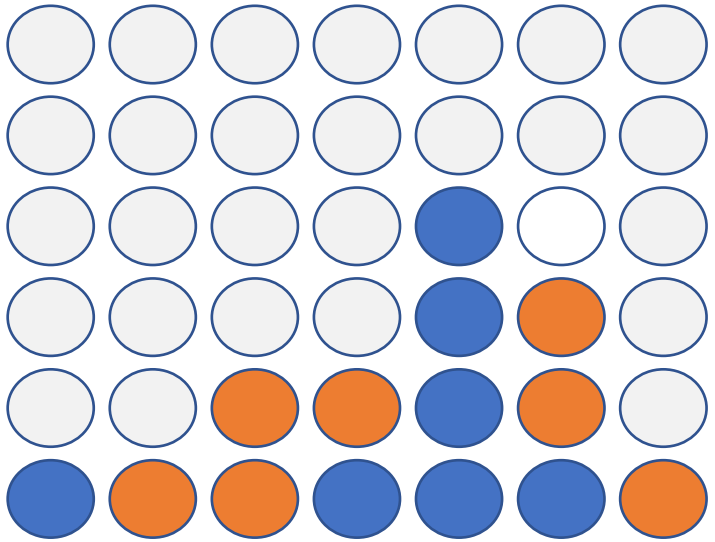


- Spielfeld besteht aus Pixeln.
  - Breite: Spaltenzahl * (2 * Radius + Abstand) + Abstand
  - Höhe: Zeilenzahl * (2 * Radius + Abstand) + Abstand + Höhe_Statuszeile
- Ermittlung der Spalte aus der Mausposition
  - Spalte = (x-Position – Radius – Abstand) / (Abstand + 2 * Radius)

Die Verwendung von Variablen ermöglicht auch andere Spielfeldgrößen und Gewinnfelder, z.B. “3 gewinnt”.

Das Spielfeld

	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



- Spielfeld wird als zweidimensionales Array abgebildet.
  (Da Python das nicht alleine kann, wird dazu die Bibliothek `numpy` verwendet.)

```
import numpy
import konstanten as ko

spielfeld = numpy.zeros((ko.ZEILEN, ko.SPALTEN))
spielfeld[0, 0] = 2
```

- Leere Felder sind auf 0 gesetzt, Felder für **Spieler 1 auf 1**, für **Spieler 2 auf 2**.

MinMax in Python

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

DocString:
Beschreibung einer Funktion und ihrer Parameter
(wird als Hilfe von der Entwicklungs-umgebung verwendet)

```
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
```

Die Bewertungsfunktion

```python
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 gewinnt; 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.GEWINNFELDER)
    wert_s1 = berechne_wert(bewertung_z1)
    bewertung_z2 = zaehle_anzahl_in_zeile(spielfeld, spiel.liefere_anderen_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
```

- Prüfe, ob Gewinnposition vorliegt
- Sonst berechne gewichtete Werte für Einer bis Dreier
- Bewerte Computer höher, um Patt zu vermeiden

Was passiert dabei?

	0	1	2	3	4	5	6
0	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

	0	1	2	3	4	5	6
0	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

	0	1	2	3	4	5	6
0	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000

	0	1	2	3	4	5	6
0	0.00000	0.00000	0.00000	2.00000	1.00000	0.00000	0.00000
1	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

- Spielfeld wird schrittweise aufgebaut. (`fuehre_naechsten_zug_aus`)
- Spielposition wird untersucht und bewertet.
- Danach wird das Spielfeld schrittweise abgebaut. (`nehme_zug_zurueck`)
- Abbruchbedingung ist Erreichen der Suchtiefe bzw. Ende der möglichen Züge.

Wie oft passiert das eigentlich?

Untersuchte Stellungen

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

- Zahlen für Computerzüge des abgebildeten Beispiels
- Suchtiefe = 6

Abhängigkeit von 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	30319	1,05
9	5	17	122707	580310	1070254	39,76
10	3	1	353667	85024	585348	22,48

- Zahlen jeweils für nahezu leeres Spielfeld (Spieler macht ersten Zug in Spalte 3, Computer macht den nächsten Zug)

Was kann man lernen?

Bildquelle: bronner-schule.de

Was kann man lernen?

Vor dem Programmieren.

- **Programmfunktionen** überlegen vor dem Programmieren
- **Algorithmus überlegen und visualisieren** vor dem Programmieren
- Komplexere Projekte brauchen eine **Unterstruktur**.
- Python hat **nützliche Bibliotheken** für viele Zwecke.

Beim Programmieren.

- **Sprechende Variablennamen, DocStrings und Kommentare** helfen, das Programm auch noch später zu verstehen.
- Die **Nutzung von definierten Konstanten anstelle von Literalen** (festen Werten im Programmcode) hilft, diese schnell anzupassen. Die Nutzung des Programms wird außerdem flexibler (”3 gewinnt”).
- Die **Nutzung von Unterprogrammen** hilft, die Übersicht zu behalten.
- **Rekursive Programme** sind gut zu lesen, aber schwierig zu debuggen.

```python
def pruefe_spalte_fuer_gewinn(spielfeld, spieler, spalte, ze
    """
    Prüft in gegebener Spalte, ob Gewinnposition für Spieler
    :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 == ze
            # Das Feld ist von diesem Spieler besetzt.
            # Zähle Feld mit.
            gesamt += 1
            if gesamt == ko.GEWINNFELDER:
                # Anzahl der Gewinnfelder ist err
                return True
        else:
            # Das nächste Feld ist von and
            # Fange wieder von vorn an
            gesamt = 0
    return False
```

Python im Alltag

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

Henriette Schulz

```
def pruefe_spalte_fuer_gewinn(spielfeld, spieler, spalte, ze
    """
    Prüft in gegebener Spalte, ob Gewinnposition für Spieler
    :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 == ze
            # Das Feld ist von diesem Spieler besetzt.
            # Zähle Feld mit.
            gesamt += 1
            if gesamt == ko.GEWINNFELDER:
                # Anzahl der Gewinnfelder ist err
                return True
        else:
            # Das nächste Feld ist von and
            # Fange wieder von vorn an
            gesamt = 0
    return False
```

Anhang

Alpha-Beta-Suche

Minerva führt Maxi in ihr riesiges Ankleidezimmer mit Regalen voller Handtaschen. Maxi soll eine Tasche geschenkt bekommen. Minerva ist aber geizig und will Maxi nur die schlechteste Tasche aus einem ausgewählten Regal schenken.

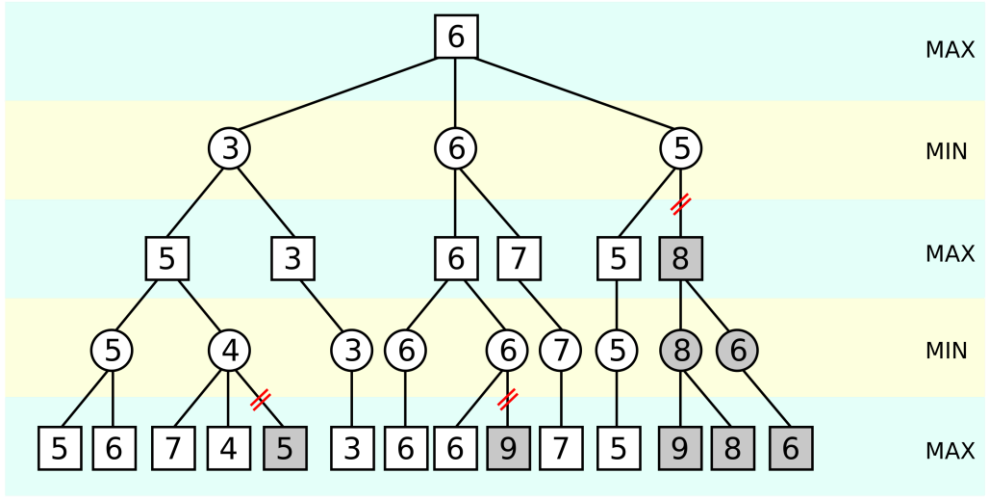
- Fall 1: Maxi darf das Regal selbst auswählen
  - Maxi wählt zunächst eins von mehreren Regalen und durchsucht es vollständig. Die schlechteste Tasche ist von Louis Vuitton – tolle Sache. Aber geht es noch besser? Schließlich hat Minerva noch eine Birkin Bag.
  - Im nächsten Regal findet Maxi sofort eine Tasche von H&M. Da braucht sie also nicht weiter zu suchen.
  - Das ist der **Alpha-Schnitt**.

- Fall 2: Minerva darf das Regal bestimmen
  - Minerva wählt ein Regal aus und findet als beste Tasche eine von Zara. Schlecht genug ;-)
  - Im nächsten Regal findet Minerva gleich eine Tasche von Gucci. Hier braucht sie nicht weiter zu suchen, denn diese Tasche oder gar noch bessere möchte sie nicht verschenken.
  - Das ist der **Beta-Schnitt**.

Bildquellen: hm.com, wikimedia.com

Beispiel nach:
Patrick Krusenotto: Funktionale Programmierung und Metaprogrammierung

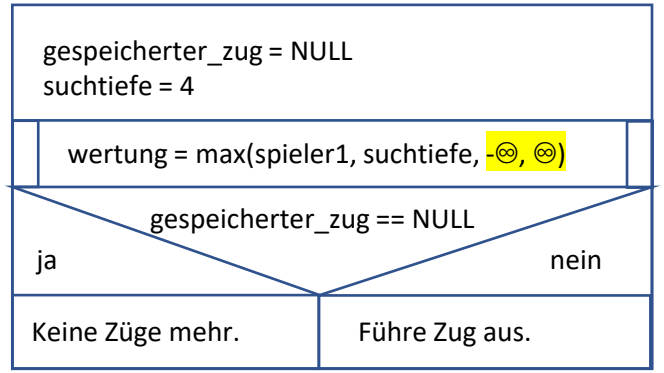
Alpha-Beta-Suche: Einschränkung des Suchraums



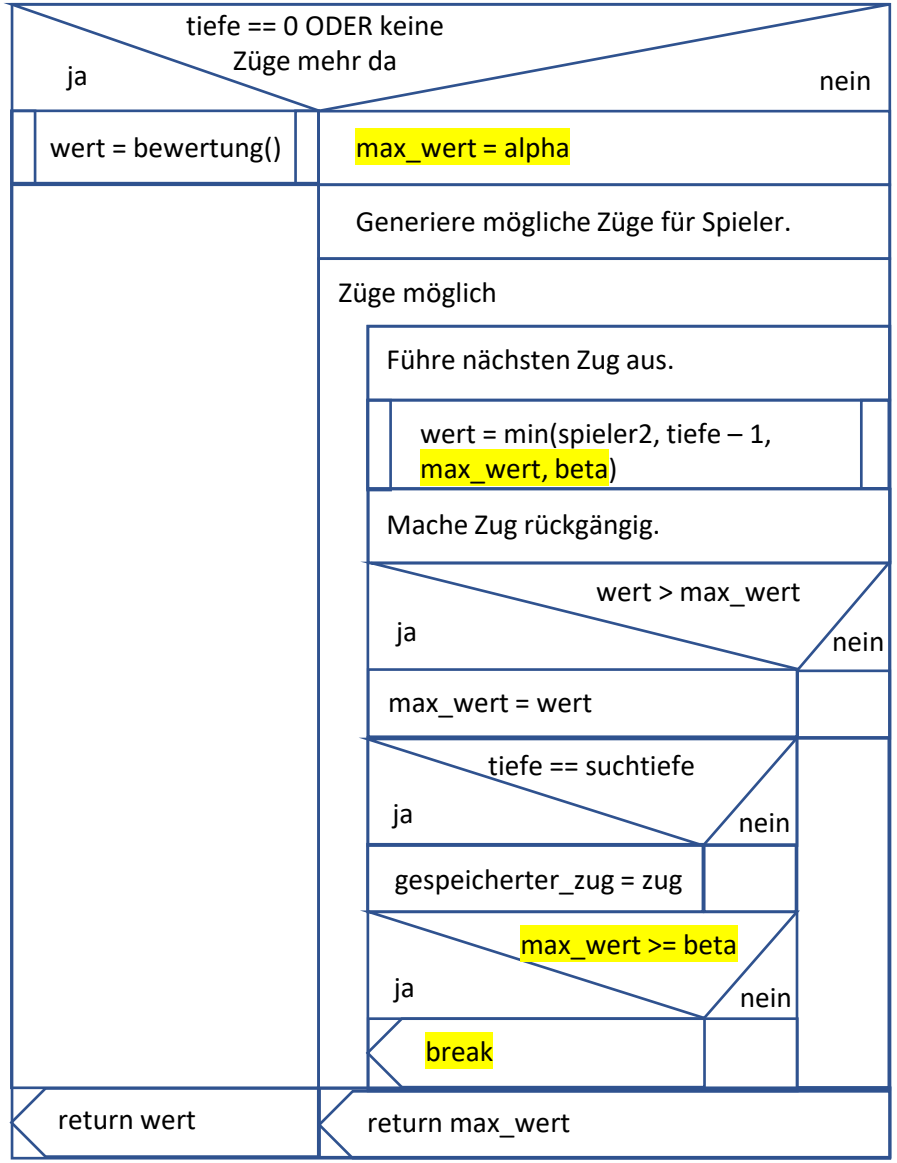
Bildquelle: Wikipedia.de, Autor: Jez9999

- Es sollen die Knoten ignoriert werden, von denen bereits feststeht, dass sie das Ergebnis nicht beeinflussen.
- α und β bilden ein Intervall, das sich von oben nach unten immer weiter zu zieht.
- Im besten Fall kann ein Spielbaum mit N Knoten bis auf $\sqrt{N}$ zusammengeschnitten werden.

Hauptprogramm



max



min