Projekt AI für 4-Gewinnt

Autoren:

* Martin Kieliger
* Sandro Luder
* Michael Schmid

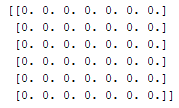
# Das Spiel

Quelle Wikipedia: «Das Spiel wird auf einem senkrecht stehenden hohlen Spielbrett gespielt, in das die Spieler abwechselnd ihre Spielsteine fallen lassen. Das Spielbrett besteht aus sieben Spalten (senkrecht) und sechs Reihen (waagerecht). Jeder Spieler besitzt 21 gleichfarbige Spielsteine. Wenn ein Spieler einen Spielstein in eine Spalte fallen lässt, besetzt dieser den untersten freien Platz der Spalte. Gewinner ist der Spieler, der es als erster schafft, vier oder mehr seiner Spielsteine waagerecht, senkrecht oder diagonal in eine Linie zu bringen. Das Spiel endet unentschieden, wenn das Spielbrett komplett gefüllt ist, ohne dass ein Spieler eine Viererlinie gebildet hat. Sollte ein Spieler das Spiel verlieren, aber alle seine Spielsteine sind miteinander verbunden, so erhält er einen Zusatzpunkt und das Spiel endet Unentschieden»

Es existiert eine optimale Strategie, bei der der anziehende Spieler immer gewinnt. Diese basiert auf 9 Regeln und ist ziemlich komplex. Quelle: Masterthesis von Victor Allis[[1]](#footnote-1). Da dieser in der kurzen Projektzeit zu schwer zu implementieren wäre, haben auf eine Implementation dieser optimalen Strategie verzichtet.

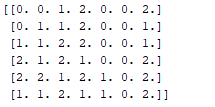
# Grundaufbau Programm

Das Spielfeld ansich ist ein simples Zweidmensionales Array, anfangs befüllt mit 0en:



Erstellt wird es mit dem Package NumPy[[2]](#footnote-2). Row- und Columcount sind standartmässig 6x7 wie beim offiziellen Spiel, können aber angepasst werden.

Ein Spielstein ist nichts anderes als eine 1 (Spieler1) oder 2 (Bot, Spieler2) im Array:



Nach dem Initialisieren des Boards geht das Spiel in den Gameloop über. Dieser läuft bis ein GameOver erreicht wird.

In der Variable turn wird der momentan aktive Spieler gespeichert. Wer den ersten Zug machen kann wird zufällig ausgewählt. Beim Mensch wartet das Programm auf einen Input. Beim Bot wird je nach Modus einfach random ein Stein gelegt oder per MinMax Verfahren (siehe unten) ein Stein gelegt.

# Modi

Aus Bastelgründen haben wir mehrere Skripts (== unterschiedliche Modi) programmiert. Nur der Modus Human vs. Ai wurde «sorgfältig» entwickelt, die restlichen Skripte sind Abwandlungen und haben deshalb überflüssigen Code etc.

## Human vs Human

Zwei Menschliche Spieler können gegeneinander antreten.

## Human vs RngAI

Ein Mensch spielt gegen einen Zufallsbot. Dieser wählt immer aus allen ValidLocations immer zufällig eine Spalte aus.

## Human vs AI

Ein Mensch spielt gegen einen Bot mit dem MinMax Verfahren (siehe unten). Mit der Depth (es wird tiefer gesucht) kann die Stärke der KI angepasst werden. Für uns 4-Gewinnt Laien scheint eine Tiefe von 3 angemessen (damit wir auch ab und zu gewinnen…). Die Berechnungszeiten werden exponentiell grösser bei zunehmender Tiefe. Folgende Zeiten wurden bei einer Proberund gemessen:

|  |  |  |
| --- | --- | --- |
| **Depth** | **Erster Zug [ms]** | **Längste KI Bedenkzeit [ms]** |
| 3 | 28 | 34 |
| 4 | 114 | 290 |
| 5 | 622 | 976 |
| 6 | 2495 | 2495 |
| 7 | 9166 | 12529 |

## AI vs AI

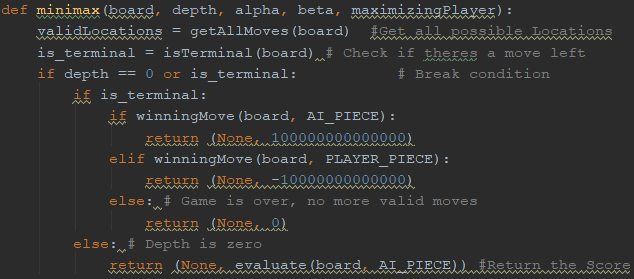
Als Spielerei. Wir haben das noch automatisiert (Datei \*\_automatisiert). Dabei spielen zwei Ais mit unterschiedlichen Tiefen und Scorecards ( Bewertungsfunktion) viele Runden (bspw 1000) gegeneinander. Ziel der Übung ist es, eine gute Scorecard und Tiefe zu finden. Ergebnisse siehe Word-Datei «BotGegenBot\_Beobachtung.docx».

## AI vs RngAI

Als Spielerei. Ein MiniMax Bot spielt gegen einen Zufallsbot. Natürlicherweise sollte der MinMax Bot bei einer brauchbaren Implementierung dabei eine massiv höhere Gewinnchance haben.

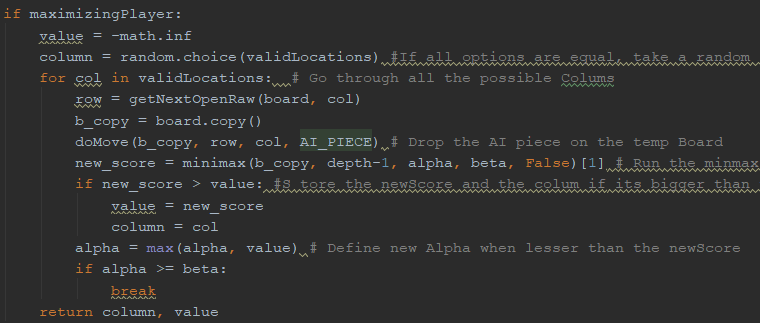
Die Ergebnisse sind in der Textdatei «AIvsRNG.txt» festgehalten. Grundsätzlich betrag die Gewinnchance gute 99.8% => Der Algorythmus funktioniert.

# MiniMax



In unserer *minimax()* Funktion wird die momentane Position (*board*), die Tiefe in welcher wir suchen wollen (*depth* -> in unserem Fall 5) *alpha* und *beta* für unser bestmöglichen Zug, respektive den bestmöglichen Zug für unseren Gegner und der boolean Wert *maximazingPlayer* welcher den nächsten Spieler bestimmt, parametrisiert.

Wir beginnen zu prüfen ob die Tiefe = 0 ist oder das Spiel an der aktuellen Position vorbei ist. Wenn ja geben wir den Satus der aktuellen Position zurück also 1 für AI Sieg, -1 für Spieler Sieg und 0 für unentschieden. Wenn Tiefe = 0 ist wird der Score zurückgegeben.



Anderseits wenn der *maximizingPlayer* am Zuge ist wird der höchstmögliche Wert in dieser Position gesucht. Somit setzen wir den maximalen Wert auf – unendlich. (value = -mat.inf). Wir loopen anschliessend durch alle möglichen Positionen. Um nun jede mögliche Position zu evaluieren wir die *minimax(b\_copy, depth-1, alpha, beta, False)* Funktion ausgeführt. Den höchsten Wert erhalten wir anschliessend mit der *max(alpha, value)* Funktion. Wenn *alpha* >= *beta* ist wird der Loop abgebrochen.

Dasselbe wird für den *minimizingPlayer* durchgeführt mit minimalem Wert auf + unendlich und den Mindestwert von *min(beta, value)*.

<https://www.youtube.com/watch?v=l-hh51ncgDI>

## Bewertungsfunktion

Ein

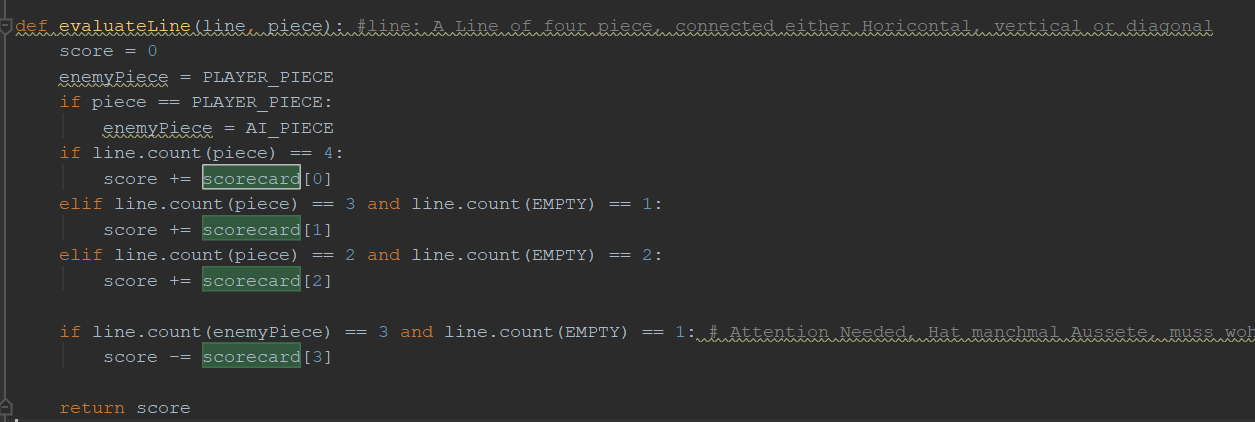


Abbildung 1 - Funktion evaluate Line

1. http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf [↑](#footnote-ref-1)
2. https://www.numpy.org/ [↑](#footnote-ref-2)