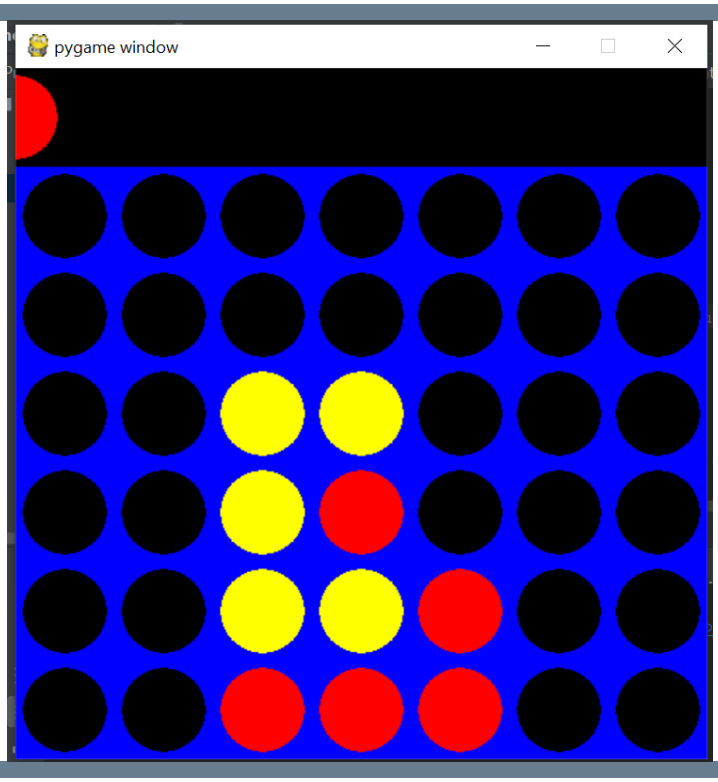

Projekt AI für 4-Gewinnt

Projektbeschrieb

Martin Kieliger (kielm1)
Sandro Luder (ludes2)
Michael Schmid (schmm11)

V1.0 / 18.06.2019

Berner Fachhochschule
Departement
Abteilung

1 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, im Kern auf dem Zugzwang Prinzip, und ist ziemlich komplex (Quelle: Masterthesis von Victor Allis[1], stolze 91 Seiten lang). Da dieser in der kurzen Projektzeit zu schwer zu implementieren wäre, haben wir auf eine Implementation dieser optimalen Strategie verzichtet und eine eigene Bewertungsfunktion (siehe Kapitel Bewertungsfunktion).

[1] http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf

2 Grundaufbau Programm

Da die Autoren zum ersten Mal mit Python programmierten, wurde in einem ersten Schritt ohne die Interface Vorlagen von Herr Eckerle (Gametree.py) gearbeitet (siehe Ordner «MainCode_ohneInterfaceVorlage»). Ein nachträgliches Einfügen des funktionierenden Codes war auf Grund Zeitmangels nicht mehr möglich (Versuch siehe Ordner VersuchMit_InterfaceVorlage). Da der Code ohne die InterfaceVorlagen gut funktioniert, ist dies nach Meinung der Autoren kein Problem.

Viele Elemente, vor allem das simple GUI, wurden mit pygame[2] (Laut Kollegen ein super Einstieg in Python mit guten Tutorials) erstellt.

Das Spielfeld an sich ist ein simples Zweidimensionales Array, zu Beginn befüllt mit 0-en:

```
[[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
```

Abbildung 1: Leeres Spielfeld

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:

```
[[0. 0. 1. 2. 0. 0. 2.]
 [0. 1. 1. 2. 0. 0. 1.]
 [1. 1. 2. 2. 0. 0. 1.]
 [2. 1. 2. 1. 0. 0. 2.]
 [2. 2. 1. 2. 1. 0. 2.]
 [1. 1. 2. 1. 1. 0. 2.]]
```

Abbildung 2: Gefülltes Spielfeld

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 Menschen 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.

[2] https://riptutorial.com/de/pygame

3 MinMax mit Alpha Beta Pruning

Anfänglich hatten wir nur ein MinMax Algorithm ohne Pruning. Nach der Besprechung am 12. Juni wurde auf Wunsch von Herr Eckerle das Ganze noch mit einem Alpha Beta Pruning erweitert.

```
def minimax(board, depth, alpha, beta, maximizingPlayer):
    validLocations = getAllMoves(board)  #Get all possible Locations
    is_terminal = isTerminal(board)# Check if theres a move left
    if depth == 0 or is_terminal:           # Break condition
        if is_terminal:
            if winningMove(board, AI_PIECE):
                return (None, 1000000000000000)
            elif winningMove(board, PLAYER_PIECE):
                return (None, -10000000000000)
            else:# Game is over, no more valid moves
                return (None, 0)
        else:# Depth is zero
            return (None, evaluate(board, AI_PIECE)) #Return the Score
```

Abbildung 3: MinMax Alpha Beta 1

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.

```
if maximizingPlayer:
    value = -math.inf
    column = random.choice(validLocations) #If all options are equal, take a random
    for col in validLocations:# Go through all the possible Colums
        row = getNextOpenRaw(board, col)
        b_copy = board.copy()
        doMove(b_copy, row, col, AI_PIECE)# Drop the AI piece on the temp Board
        new_score = minimax(b_copy, depth-1, alpha, beta, False)[1] # Run the minmax
        if new_score > value: #Store the newScore and the colum if its bigger than
            value = new_score
            column = col
        alpha = max(alpha, value)# Define new Alpha when lesser than the newScore
        if alpha >= beta:
            break
    return column, value
```

Abbildung 4: MinMax Alpha Beta 2

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, wird die *minimax(b_copy, depth-1, alpha, beta, False)* Funktion mit dekrementierter Tiefe 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)*.
Als Einstieg in das Thema diente ein Tutorial Youtube Video von Sebastian Lague[3]

[3] https://www.youtube.com/watch?v=l-hh51ncgDI

3.1 Bewertungsfunktion

Die Bewertung schaut sich pro möglichen Zug immer diverse Linien von 4 Punkten (vertikal, horizontal oder diagonal) ausgehend vom geplanten Stein aus an. Je nachdem, wieviel eigene, gegnerische oder freie Punkte verfügbar sind, wird ein Score vergeben. Der Score ist zum besseren Überblick und zur schnellen Anpassung als Array «Scorecard» gespeichert (siehe Abbildung).

- Für 4 eigene Steine: Extrem viele (1000) Punkte, da dies ein Sieg bedeutet.
- Für 3 eigene Steine und 1 freier: Mittelmässig (5) Punkte, da nahe am Sieg.
- Für 2 eigene Steine und 2 freie: nur wenige (3) Punkte.
- Für 3 gegnerische und 1 freier: Viele negative Punkte (7), da der Gegner hier behindert werden muss.

Im Modus «HumanvsAI» und «AIvsAI» haben wir mit verschiedenen Scorecards herumgespielt, die Einstellung von [1000, 5,3,7] stellte sich dabei als kompetitiv heraus.

```
# The score for:
# - [0]: line of 4 own Pieces
# - [1]: line of 3 own Pieces and 1 empty
# - [2]: line of 2 own Pieces and 2 empty
# - [3]: line of 3 enemy Pieces and 1 empty (counts negative)

scorecard = [1000, 5, 3, 7]
```

Abbildung 5: Scorecard

```
def evaluateLine(line, piece): #line: A Line of four piece, connected either Horicontal, vertical or diagonal
    score = 0
    enemyPiece = PLAYER_PIECE
    if piece == PLAYER_PIECE:
        enemyPiece = AI_PIECE
    if line.count(piece) == 4:
        score += scorecard[0]
    elif line.count(piece) == 3 and line.count(EMPTY) == 1:
        score += scorecard[1]
    elif line.count(piece) == 2 and line.count(EMPTY) == 2:
        score += scorecard[2]

    if line.count(enemyPiece) == 3 and line.count(EMPTY) == 1: # Attention Needed, Hat manchmal Aussete, muss wob
        score -= scorecard[3]

    return score
```

Abbildung 6 - Funktion evaluate Line

4 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.

4.1 Human vs Human

Zwei Menschliche Spieler können gegeneinander antreten.

4.2 Human vs RngAI

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

4.3 Human vs AI

Ein Mensch spielt gegen ein 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 Proberunde gemessen:

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

Tabelle 1: Human vs AI

4.4 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».

4.5 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 Algorithmus funktioniert.