# Bericht Implementation Hex Game

Marco Ging

# Einleitung

Für das Modul Spieltheorie (BTI7501p) wurde gefordert, ein Hex-Board-Spiel zu entwickeln mit folgenden Anforderungen.

- Realisierung des Spiels Hex, vorzugsweise in Python oder Java, Anforderungen: Dynamische wählbare Grösse des Spielbretts, Abgabe des Codes in einem File (!), Startbar aus der Kommandozeile, kurze Installationsanleitung.
- Implementieren eines Computer Gegners welcher den Minimax Algorithmus anwendet
- Implementieren eines Computer Gegners welcher den Alpha-Beta-Pruning Algorithmus anwendet
- Implementation des Spieles durch eine Klasse
- Heuristische Bewertungsfunktion finden
- Möglichkeit Mensch vs. Mensch, Mensch vs. Computer, Computer vs. Computer zu spielen.
- Bestimmung welcher Spieler eine Gewinnstrategie hat.
- Bestimmung der maximalen Spielfeldgrösse, mit der bestimmt werden kann, welcher Spieler eine Gewinnstrategie aufweist.
- Erweiterung des Spiels mit neuen Spielregeln
    - nicht-deterministische Zugauswahl
    - Switchen des Spielbretts
    - Tausch von zwei unterschiedlich farbigen Spielsteinen, usw.
- Darstellung der implementierten Verfahren und der Ergebnisse in einem Bericht (was und wie wurde etwas implementiert)

# Installationsanleitung

Um das Spiel starten zu können, werden 2 Bibliotheken benötigt, welche zuvor noch mit pip3 installiert werden müssen.

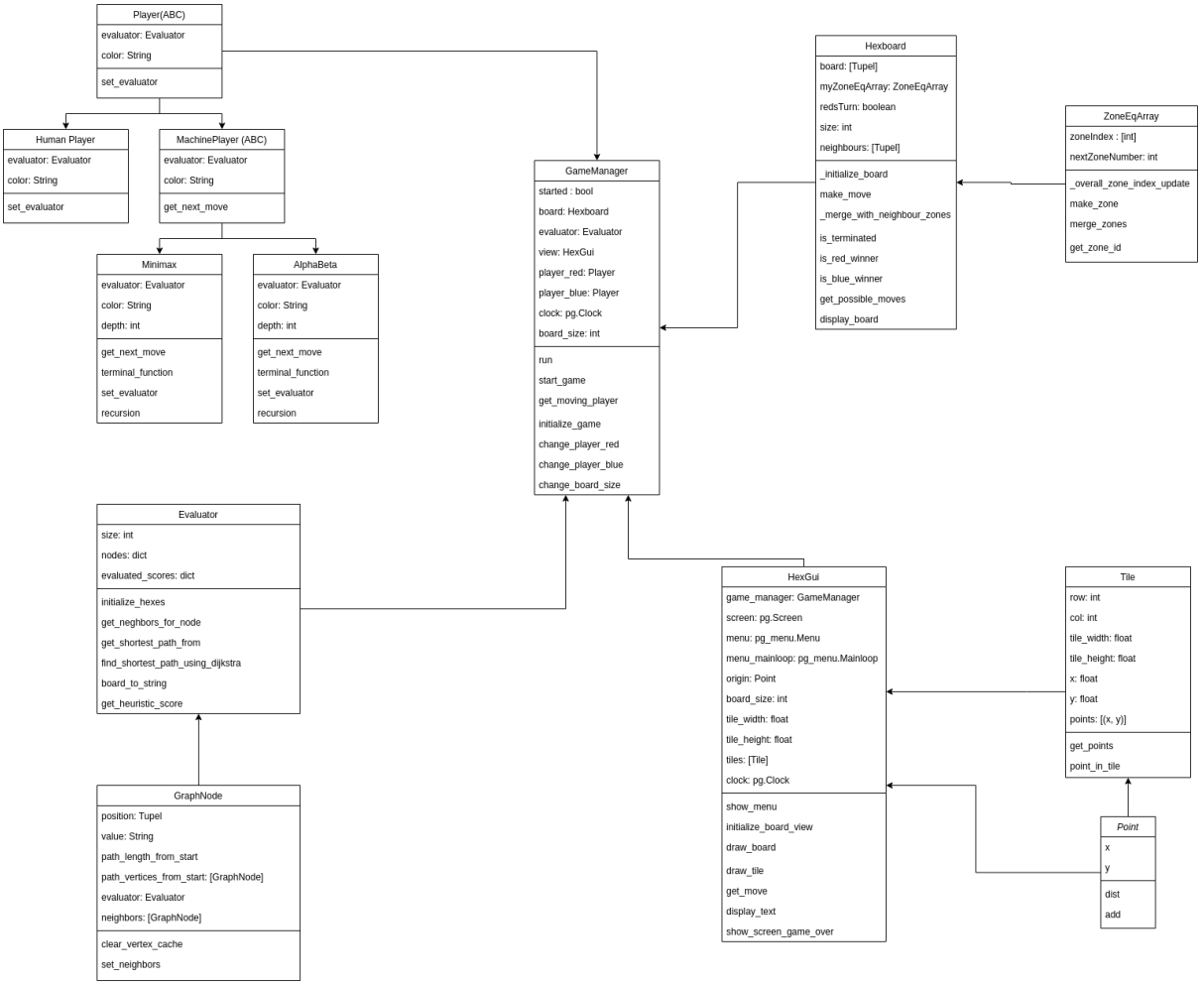
```
pip3 install pygame pygame_menu
```

Anschliessend kann das programm aus der konsole mit folgendem Befehl ausführen

```
python3 ./HexBoardGame.py
```

# Implementation

## Klassendiagramm



# Aufbau

Für die Implementation wurde versucht, das Model - View - Control Modell umzusetzen. Dazu dient die Klasse GameManager als Controller, die Klasse HexGui als View und bei den Models gibt es die Spieler, Gameboard und der Evaluator. Die Implementation des Spiels wurde mit Hilfe der Bibliothek Pygame realisiert, welche für das Erstellen eines GUI's sehr hilfreich war.

# Controller

Der Controller in unserem Fall die Klasse GameManager ist zuständig für die Kommunikation zwischen den Models und der View. Er weiss wann das Spiel gestartet ist und welche Spieler teilnehmen (menschlicher Spieler, welcher Algorithmus). Er hat die Aufgabe die Instanzen der Modelle zu initialisieren und starten den mainloop des Spieles.

## Ablauf des Spieles.

In einer Main-Methode wird der GameManager initialisiert und die "run" Methode aufgerufen. Diese überprüft, ob das Spiel bereits gestartet wurde und wenn nicht, wird ein Startmenü angezeigt. Wenn die Spieler und die Spielfeldgrösse ausgewählt wurde, initialisiert der GameManager die Spieler und das Spielbrett und startet das Spiel in einem mainloop. Der Mainloop überprüft, welcher Spieler am Zug ist und wartet auf einen Userinput bei einem menschlichen Spieler oder ruft die methode get_next_move bei einem maschinen Spieler auf. Wenn das Spiel beendet wurde, ruft der GameManger die View game over auf, wodurch ein Text mit dem Gewinner angezeigt wird.

# View

Die view Klasse HexGui dient dazu, das Spiel in einem GUI darzustellen. Für die Implementation wurde die Libary pygame und pygame_menu verwendet.

## Menu

Das Menu zu beginn wurde nicht durch ein MVC modell implementiert, da die Libary pygame_menu ein solches Modell nicht direkt supportet. Daher wird das Menu direkt von der View kontrolliert und Änderungen der Spieler oder der Spielfeldgrösse dem Controller übergeben (methoden change_player und change_board_size). Mit dem klick auf Play wird das spiel vom Controller initialisiert und die Game board view dargestellt.

## Game board

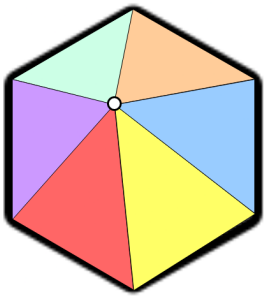
Das Game board wurde “dynamisch” implementiert, so dass die Grösse der Hexagons auf dem Spielfeld relativ zur Spielfeldgrösse sind. Die Grösse wird wie folgt ermittelt:

```
self.tile_width = ((WINDOW_WIDTH - 100) / (self.board_size + (self.board_size - 1) * 0.5))
self.tile_height = 2 * self.tile_width / sqrt(3)
```

Für die Spielerfarben am Rand wurden vier Dreiecke gezeichnet welche zusammen einen Rombus ergeben und die Hexagons umschliesst.

Für das Darstellen des Spielfelds wurden die einzelnen Hexagons in eine eigene Klasse abgebildet namens Tile. Jedes Tile besitzt die Information über die Position auf dem Spielfeld (Zeile Spalte) und die Information der Punkte auf dem Screen (x, y - Koordinaten der einzelnen Ecken). Darüber hinaus haben sie eine Funktion namens point_in_tile. Diese Funktion wird verwendet, wenn der Spieler auf dem Screen einen Mausklick ausführt. Durch den Mausklick wird die Position vom Controller an die View weitergegeben und dann für jedes Hexagon (Tile) ermittelt, ob diese Position auf ein Hexagon trifft.

Um herauszufinden, ob die Position, respektive der Punkt, sich in einem Hexagon befindet, wird die Fläche der 6 Dreiecke berechnet, welche resultieren, wenn man den Punkt und jeweils die Ecken der Seiten des Hexons nimmt. Wenn die Fläche dann der Fläche des Hexagons entspricht, ist dieser Punkt in dem Hexagon.



Die letzte Aufgabe der view ist das Darstellen des game over screen. Dabei wird ein Schriftzug Game Over über das Spielfeld gezeichnet und der Gewinner benannt.

# Model

In den einzelnen Models befindet sich die Logik der einzelnen Komponenten.

## HexBoard

Das Hexboard wurde bereits vorgegeben und enthält das board, welches ein Liste von Tupeln besitzt. In den einzelnen Tupel wird der Wert (“R”, “B”, “X” ) und ein Zonenindex gespeichert. Dieser Zonenindex wird verwendet um zu überprüfen ob das Spiel bereits beendet wurde. Das Board weiss wer am Zug ist mit Hilfe des Boolean redsTurn und kennt die Spielfeldgrösse.

Das initialisieren des Boards erfolgt mit einem Rahmen um das Spielfeld mit dem Wert “X” und einem entsprechenden Zonenindex. Der Rest des Boards wird nicht gesetzt.

Bei einem Zug (make move) wird an der Position wo der Zug stattfindet erst ein Tupel (Wert, -1) erstellt. Anschliessend wird mit der Funktion _merge_with_neighbour_zones die entsprechende Zone gesetzt.

Dies funktioniert wie folgt:
Es wird erst überprüft, ob ein Nachbar den gleichen Wert besitzt wie der aktuell gemachte Zug. Wenn dies eintrifft und die Zone weiterhin “-1” ist, wird die Zone des Nachbarns mit dem gleichen Wert übernommen. Wenn die Zone des aktuellen Zugs nicht “-1” ist und es einen Nachbarn gibt mit dem gleichen Wert, jedoch einem anderen Zonenindex, werden die Zonen gemerged.

Da zu beginn jede Seite einen anderen Zonenindex besitzt (1 - 4) und verbundene Zonen den gleichen Wert erhalten, kann während dem Spiel jeweils überprüft werden ob der Rand oben und unten respektive links und rechts der gleiche Zonenindex besteht. Wenn dies der Fall ist, ist das Spiel beendet und es gibt einen Gewinner.

## Spieler

In unserem Spiel gibt es 2 Arten von Spielern. Nämlich die Human players und die Machine Players. Bei dem Human player wird ein Input des Users verlangt (klick auf ein leeres Hexagon). Bei einem Machine Player wird im Hintergrund ein Algorithmus angewandt, um den besten Zug zu spielen. In dieser Version des Spieles gibt es aktuell 2 Algorithmen nämlich den Minimax und den Alpha Beta Algorithmus.

## Minimax

Der Kontroller initialisiert den Minimax-Spieler mit einer Tiefe, einem Evaluator und einem Wert, also Rot oder Blau. Der Kontroller kann, wenn der Minimax-Spieler am Zug ist seine Funktion get_next_move aufrufen wobei er das Hexboard übergibt.

Die Funktion beginnt mit einer Rekursion mit den Werten (HexBoard, maxplayer = true und der Tiefe). In der Rekursion wird überprüft, ob das Spiel bereits beendet wurde. Wenn dies der Fall ist, wird entweder der Wert + oder - Infinity zurückgegeben, je nachdem ob der Minimax-Spieler gewonnen oder verloren hat. Wenn das Spiel nicht beendet wurde, werden die möglichen Züge vom Hexboard genommen. Für jeden möglichen Zug wird ein neues HexBoard erstellt und die Rekursion wiederholt mit einer Tiefe welche eins kleiner ist und der maxplayer invertiert wird. Dies wird wiederholt, bis die Tiefe auf 0 ist. Wenn dies eintrifft und das Spiel noch nicht entschieden wurde, wird ein Heuristischer Score ermittelt, welcher in der Evaluator Klasse herausgefunden wird. Ich werde dies genauer in der Evaluator Klasse vertiefen.
Für jeden Zug bekommen wir dann einen Score. Wenn der Maxspieler den besten Zug auswählt, entscheidet er sich für den höchsten Score und beim Min Player den kleinsten. Somit kann ermittelt werden, welcher Zug der beste ist.

## Alpha Beta Pruning

Der Alpha Beta Pruning Algorithmus ist eine Verbesserung des Minimax Algorithmus. Er verringert die Anzahl der zu untersuchenden Züge, indem er bestimmte Teile des Baums eliminiert, die keinen Einfluss auf das endgültige Ergebnis haben. Dadurch wird die Effizienz des Algorithmus erhöht.
Um Alpha-Beta-Pruning zu implementieren, wird der Minimax-Algorithmus erweitert. Neben den Parametern für das Hexboard, den maxplayer und die Tiefe werden zwei weitere Parameter, alpha und beta, hinzugefügt. Diese Parameter repräsentieren die Grenzen für den besten Zug, der von einem Spieler gefunden werden kann.
In der rekursiven Funktion wird nach dem Überprüfen, ob das Spiel beendet ist, eine zusätzliche Überprüfung hinzugefügt, ob alpha größer oder gleich beta ist. Wenn dies der Fall ist, wird die Rekursion abgebrochen und der aktuelle Score zurückgegeben. Dies bedeutet, dass dieser Teil des Baums nicht weiter untersucht werden muss.

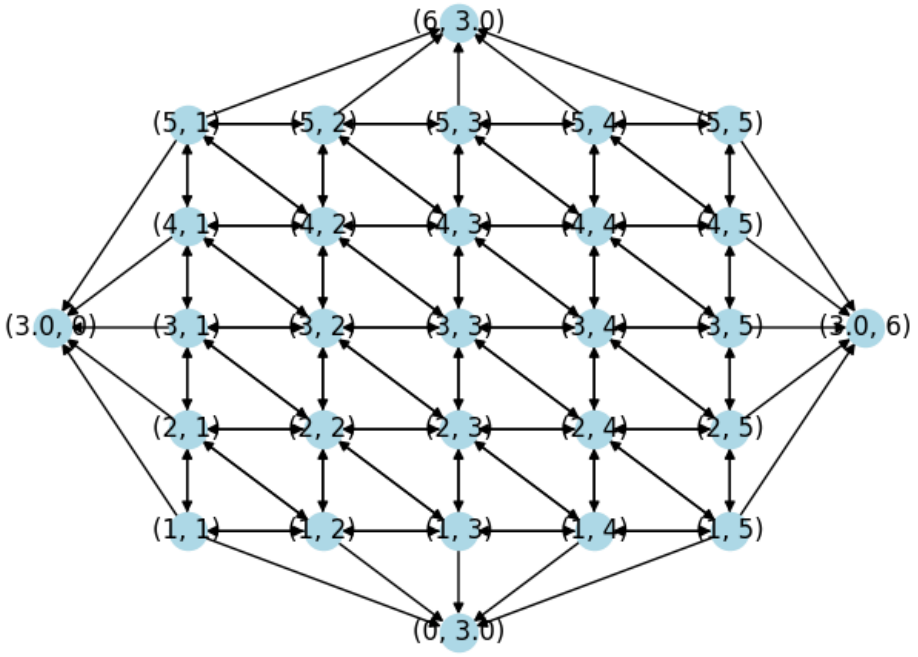
Zeitanalyse Minimax alphabeta

| Algorithmus / Spielfeldgrösse | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|
| Minimax 1 | 0.05 | 0.28 | 0.79 | 2.02 | 4.49 | 18.65 | 19.1 |
| Alphabeta 1 | 0.01 | 0.02 | 0.03 | 0.03 | 0.07 | 0.12 | 0.19 |
| Minimax 2 | 0.73 | 3.95 | 16.2 | 53.46 | 157.89 | - | - |
| Alphabeta 2 | 0.23 | 0.24 | 0.68 | 1.67 | 3.85 | 8.36 | 19.38 |
| Minimax 3 | 7.01 | 59.43 | 351 | - | - | - | - |
| Alphabeta 3 | 0.6 | 3.23 | 13.97 | 48.68 | 142.15 | - | - |
| Minimax 4 | - | - | - | - | - | - | - |
| Alphabeta 4 | 3.05 | 26.55 | 161.65 | 741.8 | - | - | - |

In der Zeitanalyse sieht man sehr gut, dass sich die Reaktionszeiten des Algorithmus massiv verbessert haben mit dem Alpha Beta-Pruning. Wenn wir uns zum Beispiel das Spielfeld mit 5 Reihen und Zeilen ansehen, sehen wir, dass der Minimax Algorithmus ca 1 Minute dauert, wobei der verbesserte Algorithmus bereits nach 3 Sekunden einen Zug findet. Ich habe abgebrochen, wenn der Algorithmus mehr als eine Minute dauert. Wir sehen, dass der Algorithmus jedoch bereits bei der Grösse 8 nicht mehr mal die Tiefe 3 unter einer Minute schafft. Somit ist eine gute Zugauswahl bei Standardgrössen 11 oder höher nicht möglich.

## Evaluator

Der Evaluator ist dazu da, eine beliebige Spielposition mit Heuristiken bewerten zu können. Zu Beginn erhält der Evaluator eine Board-Konfiguration und kreiert daraus einen Graphen mit Nodes, die seine Nachbarn kennen. Beispiel bei einem 5 x 5 board.




Die Nodes kennen ihre Position und Farbe. Mithilfe des Dijkstra-Algorithmus kann nun die Länge des kürzesten Pfades bestimmt werden, um von dem äusseren oberen zum äusseren unteren oder vom äusseren linken zum äusseren rechten Knoten zu gelangen.

Die Bewertung basiert auf der Differenz zwischen der Länge des Pfads des Gegners und der Länge des eigenen kürzesten Pfads. Wenn der Gegner einen längeren Pfad hat, erhält man eine positive Bewertung, was auf einen Vorteil hinweist.

Um die Berechnungen nicht mehrfach für dieselbe Spielfeldkonfiguration durchführen zu müssen, speichert der Evaluator die Konfigurationen zusammen mit ihrem heuristischen Score in einem Dictionary. Dadurch kann der Evaluator bereits bewertete Positionen schnell abrufen und wiederverwenden.

Der Evaluator ermöglicht somit eine effiziente Bewertung der Spielpositionen, um die bestmöglichen Züge zu ermitteln und die Entscheidungsfindung des AI-Spielers zu unterstützen.