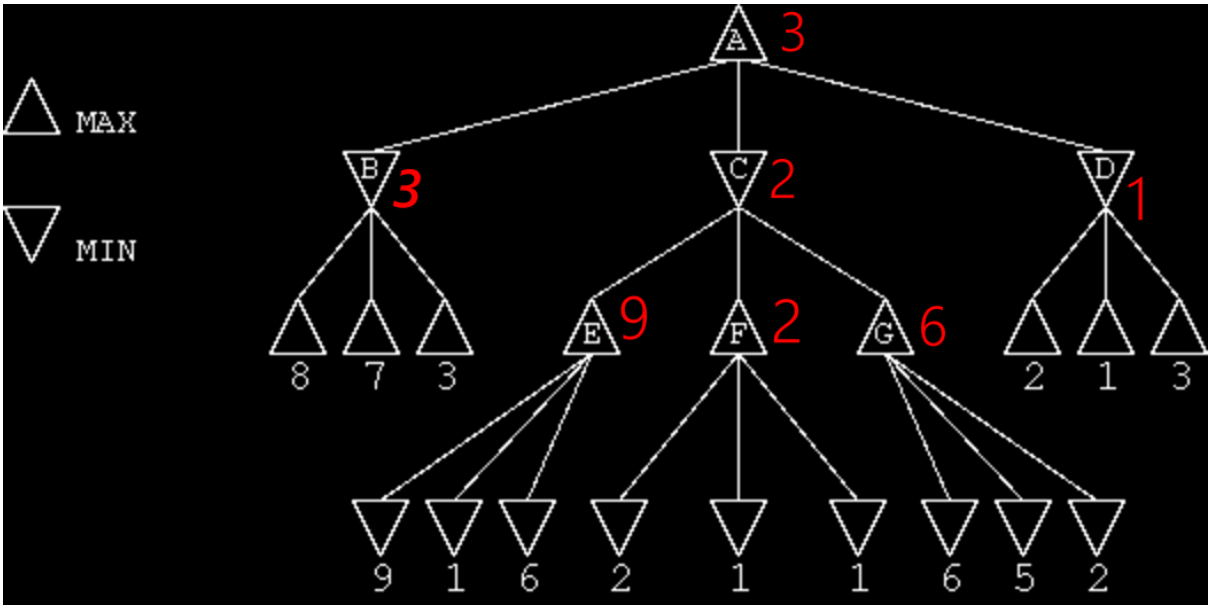
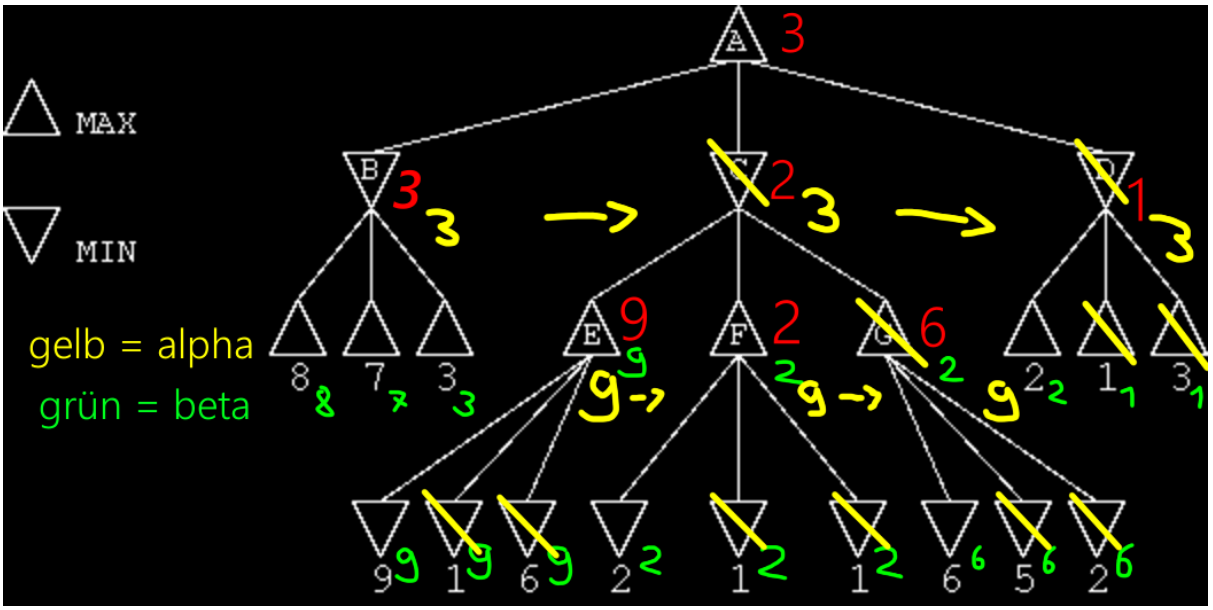
Nr.1

1.

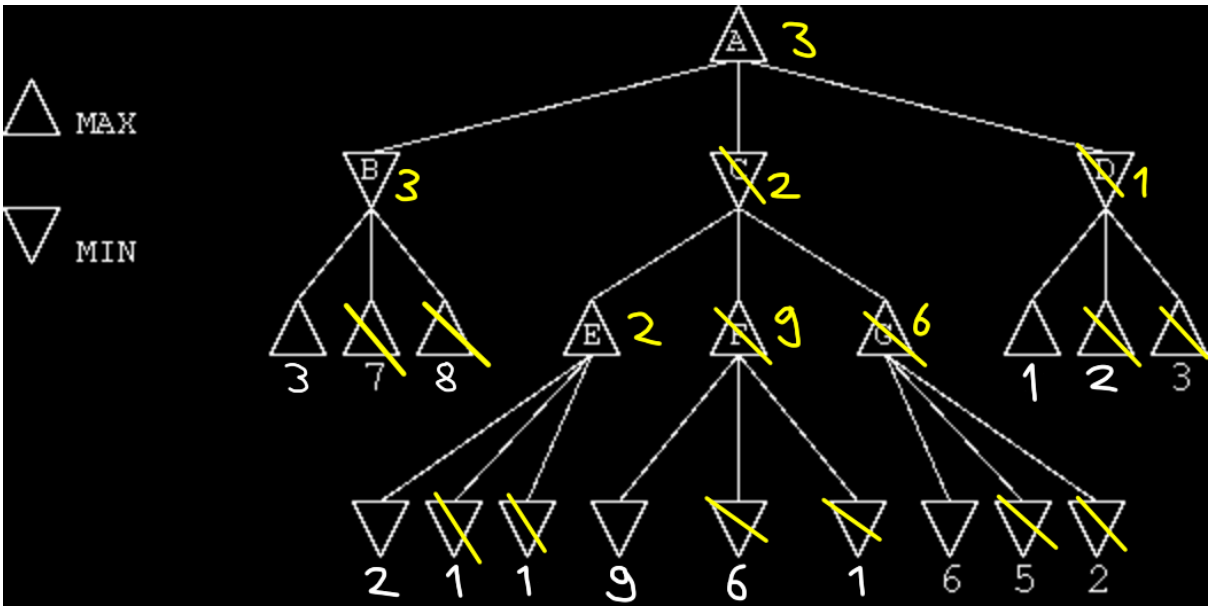


2.



Die gelben Striche stehen für Pruning.

3.



Wenn man die Knoten jeweils nach den größten/kleinsten Werten sortiert (größte zuerst für den max-Spielerr, kleinste Werte zuerst für den Min-Spieler), kann man sehr viele Teilbäume überspringen.

Nr. 2

3.

In meiner Implementierung besteht ein Spielfeld, welches nur oben in der mitte ein “O” hat. Der Rest ist frei. Hier gibt es sehr viele verschiedene Spielabläufe. Die Anzahl der Knoten im Spielbaum wächst exponentiell mit der Anzahl der freien Felder im Tic Tac Toe Feld.

Wenn wir also wie hier ein fast leeres Feld haben, werden sehr viele Knoten berechnet. In diesem Fall sind es ohne Pruning 34312 berechnete Knoten. Mit Pruning wurden nur 8942 Knoten berechnet und Pruning wurde 11967 Mal durchgeführt. Ohne Pruning wurden 3.8 Mal mehr Knoten berechnet.

Wenn wir hier als Beispiel ein Gerät haben, dessen Speicher klein ist oder dessen Rechengeschwindigkeit langsam ist, muss man eventuell lange auf den Zug des Computer-Gegners warten.

Nr. 3

beide Funktionen in einer Funktion:

```
def minimax(state, isMaximizing):
        // wenn Endzustand: Score zurückgeben
        if Terminal-Test(state): return Utility(state)
        // Minus oder Plus je nach maximieren oder minimieren
        if (isMaximising) best = -INF
        else best = INF
        // für alle Folgeknoten und Endknoten
        for (a, s) in Successors(state):
                // nächsten Knoten (maximieren oder minimieren abwechselnd)
                naechster_knoten = minimax(s, !isMaximizing)
                // den höchsten oder niedrigsten Wert für den nächsten Knoten nehmen
                best = max(best, val) if isMaximizing else min(best, val)
        return best
```

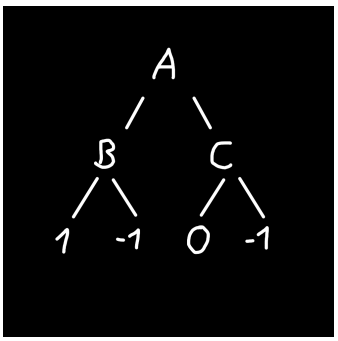
Diese Funktion verkürzt mit Nullsummenlogik

```
def minimax(state):
        // wenn Endzustand: Score zurückgeben
        if Terminal_Test(state):
                return Utility(state)

        besterScore = -INF     // Startwert ganz tief

        for (a, s) in Successors(state):
                gegnerScore = minimax(s)     // Gegner score berechnen
                score = -gegnerScore        // Maximierer Wert ist -Wert des Gegners
                besterScore = max(besterScore, score) // Bestes für den Maximierer wählen
        return besterScore
```

Überlegen Sie sich einen Beispielbaum und zeigen Sie anhand dessen die Bewertung durch den Minimax-Algorithmus und durch Ihren vereinfachten Algorithmus.



Minimax (A ist max, B und C sind min):
B: min(1, -1) = -1
C: min(0, -1) = -1
A: max(-1, -1) = -1

Nullsummenregel Verkürzung:
B: max(-1, 1) = 1
C: max(0, 1) = 1
(bei B und C sind die Werte umgedreht, da B und C die Züge des min-Spieler sind)
A: max(-1, -1) = -1

Nr. 4

Geben Sie die Werte der Evaluierungsfunktion für sechs verschiedene Spielzustände an (3 Endzustände, 3 Zwischenzustände).

$$Eval(s)=3X_2(s)+X_1(s)-(3O_2(s)+O_1(s))$$

Zustand 1
(- - -)
(X - -)
(- - -)
3*0+1-(3*0+0) = 1

Zustand 2
(X - -)
(X O -)
(- - -)
3*1+1-(3*0+2) = 3

Zustand 3
(X - -)
(X O -)
(O - -)
3*0+1-(3*1+1) = 1-5 = -4

Endzustände
1
(X X X)
(O O -)
(- - -)
Utility Funktion: 1 (X gewinnt)

2
(X X O)
(O O X)
(O - -)
Utility Funktion: -1 (O gewinnt)

3

(X O X)
(X O O)
(O X X)
Utility Funktion: 0 (unentschieden)

Begründen Sie, warum diese Evaluierungsfunktion im Zusammenhang mit Tic-Tac-Toe sinnvoll sein kann.

Die Evaluation funktion ist sehr hilfreich, da sie den Spielstand schneller bewerten kann als die vorher implementierte Utility Funktion in Aufgabe 2. Meine Funktion hat für jeden Zustand bewertet, ob ein Spieler gewinnt. Aber diese evaluate-Funktion guckt nicht nur nach dem Gewinnen, sondern sie kann auch Zustände bewerten, bei denen noch kein Spieler gewonnen hat. So musst man den Algorithmus nicht beenden, um einen Spielstand bewerten zu können. Wie ich in Aufgabe 2 herausgefunden habe, kann es vorallem zum Anfang des Spiels tausende Möglichkeiten geben, wie das Spiel weiter verlaufen kann.

Nr. 5