HTL Anichstraße
Abteilung Elektronik & techn. Informatik

Wintersemester 2019
Sommersemester 2020

Digitale Mitschrift, Version 1.1*

C - TicTacToe

Computerspiel in ASCII-text, mit Computergegner

Simon Graber, Engelbert Gruber

14. April 2020

*Version 1.1: Korrekturen, Erweiterungen

Inhaltsverzeichnis

Kapitel 1.

Dokumentation

1.1. Einführung

Mein TicTacToe-Spiel aus dem FSST-unterricht soll verbessert werden und unter anderem einen Computergegner enthalten.
Besonderer Dank gilt Engelbert Gruber, welcher meinen Code durchgesehen hat und mir zahlreiche Verbesserungsvorschläge, Ratschläge und Hinweise gab.

1.1.1. Verwendete Materialien

- Texteditor
- C-Compiler `gcc`
- Papier, Stift

1.2. Konzept

Das Ziel war es, Code zu schreiben, der möglichst wenig statisch und repetitiv (copy & paste) ist. Rechenarbeit soll dem Computer überlassen werden. Nach dem Motto »Immer, wenn ein Codesegment kopiert und nichts bis wenig daran geändert wird, ist etwas schief gelaufen.« wurde viel Funktionalität in abstraktere, dafür mächtigere Konstrukte ausgelagert. C ist nicht geeignet, auf dieser Abstraktionsebene zu operieren, Funktionen wie `col_sum, row_sum, trace` und `anti_trace` könnten in höheren Sprachen zu einer Funktion verallgemeinert werden. C bietet diese Möglichkeit nicht. So werden beispielsweise bei der Prüfung, ob das Spiel zu Ende ist[i] nicht alle Fälle einzeln abgeklappert, sondern der Absolutwert der Zeilen-, Spalten- und Diagonalensummen berechnet. Ist dieser 3, so braucht man nur das Vorzeichen der Summe zu wissen, um den Sieger zu bestimmen. [ii]
Geht man so vor, muss man zuerst abstrakte mathematische Funktionen, welche es in der Form in C nicht gibt, einbinden oder definieren. Dies wurde am Anfang des Programmes getan.

[i] Das Spiel ist zu Ende, wenn sich 3 gleiche Zeichen in einer Reihe, Spalte oder Diagonale befinden.
[ii] Mehr dazu später.

```
+---+-------+  +---+-------+  +---+-------+  +---+-------+
| 1 |       |  | 1 |   O   |  | 1 | O O X |  | 1 | O O X |
| 2 |       |  | 2 |   X   |  | 2 |   X   |  | 2 | O X   |
| 3 |       |  | 3 |       |  | 3 |       |  | 3 | X     |
+---+-------+  +---+-------+  +---+-------+  +---+-------+
    | 1 2 3 |      | 1 2 3 |      | 1 2 3 |    X | 1 2 3 |
    +-------+      +-------+      +-------+      +-------+
O(Simon)>12    O(Simon)>11    O(Simon)>21    Computer hat gewonnen!
```

Abbildung 1.1.: Spielverlauf in ASCII-darstellung

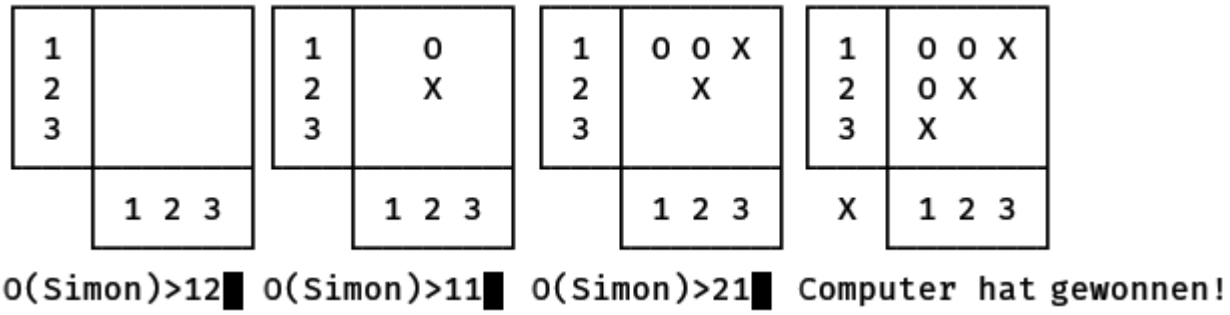

Abbildung 1.2.: Spielverlauf in extended-ASCII-darstellung

1.2.1. Spieler

Der Spieler wird zu Beginn nach seinem Namen gefragt, dann startet das Spiel. Spieler erhalten zwei feste Zahlen, zwischen denen zu jedem Zeitpunkt ein eindeutiger, linearer Zusammenhang besteht. Der Grund, zwei Zahlen zu verwenden ist jener, dass einerseits der Sieg direkt aus dem Spielfeld errechnet werden soll (hier wird `r` verwendet, siehe auch Abb. 1.3), andererseits die Spielernamen und Zeichen(O, X) im Eingabe-prompt angezeigt werden sollen(hier wird `cpn` verwendet). Eine Gegenüberstellung der Variablen findet man in Tab. 1.1.

1.2.2. Spielbrett

Das Spielbrett ist quadratisch und hat 9 Felder. Um die Feststellung des Gewinns und die Berechnung der Züge des Computers zu erleichtern, sind sie wie Elemente einer 3×3-matrix beschriftet.

r	cpn	Symbol
−1	0	O
+1	1	X

Tabelle 1.1.: Erklärung der verwendeten Zählvariablen

```
+---+-------+  +---+---------+  +---+---------+  +---+----------+
| 1 | 0 0 0 |  | 1 | 0 -1 0  |  | 1 |-1 -1 +1 |  | 1 | -1 -1 +1 |
| 2 | 0 0 0 |  | 2 | 0 +1 0  |  | 2 | 0 +1  0 |  | 2 | -1 +1  0 |
| 3 | 0 0 0 |  | 3 | 0  0 0  |  | 3 | 0  0  0 |  | 3 | +1  0  0 |
+---+-------+  +---+---------+  +---+---------+  +---+----------+
    | 1 2 3 |      | 1  2 3  |      | 1  2  3 |   +1 |  1  2  3 |
    +-------+      +---------+      +---------+      +----------+
O(Simon)>12    O(Simon)>11    O(Simon)>21    Computer hat gewonnen!
```

Abbildung 1.3.: Tatsächliche Inhalte der `board`-matrix während des Spiels.

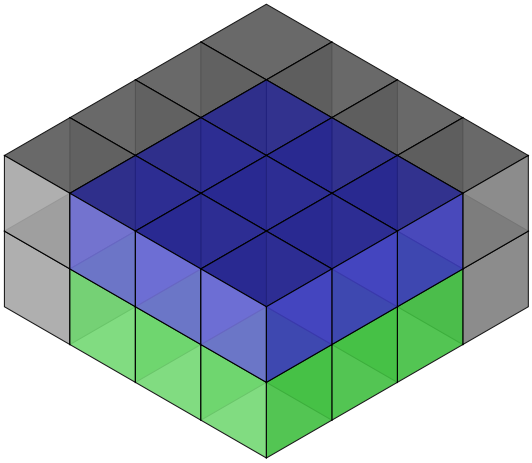

Abbildung 1.4.: Vorstellung der Spielbrettmatrix, grün markiert `board[i][j][0]` und blau markiert `board[i][j][1]`, der graue Bereich wird bislang nicht verwendet.

$$\texttt{board[i][j][0]} = \left(a_{ij}\right) = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, \quad i,j \in \{1,2,3\} \tag{1.1}$$

Freie Felder werden in der Matrix durch die Zahl 0 repräsentiert, der Spieler X durch +1, der Spieler O durch −1. Im Code wird die Spielbrettmatrix als dreidimensionales Feld `int board[4][4][2]` umgesetzt, die Variable des momentane Spielers wird `r` genannt. `r` kann nur 0, +1 oder −1 sein. Die 9 Felder sind in `board[1-3][1-3][0]` untergebracht, die anderen Felder der Matrix haben einen anderen, oder gar keinen Zweck. Man kann sich `board` als 4×4×2-quader vorstellen. (Siehe Abb. 1.4) Die Felder des Spielbretts werden in der unteren Ebene in der unteren Ebene (`board[][][0]`, grün) untergebracht. In der oberen Ebene (`board[][][1]`, blau) wird die Bewertung des Computers für jedes einzelne Feld darunter abgelegt. Die Grauen Felder sind momentan nutzlos sie dienen nur als Platzhalter, waren früher als Speicherplatz für Summen gedacht.

Return	Erklärung
0	kein Gewinner
+1	Spieler +1=X gewinnt
−1	Spieler −1=O gewinnt
61	Unentschieden[iii]

Tabelle 1.2.: Erklärung der Rückgabewerte der Funktion `wincheck()`

1.3. Gewinn-Feststellung

Wie bereits kurz erwähnt, wird im Programm bestimmt, ob ein Sieg vorliegt, oder nicht. (Siehe `wincheck` im Quellcode.) Damit ein Spieler gewinnen kann muss er entweder eine Reihe, Spalte oder Diagonale »voll« haben. D.h. die Summe aller 3 Elemente in dieser R./S./D. ist $\pm 3 \Leftrightarrow |\Sigma| = 3$ Es genügt auf diesen Fall zu prüfen und, falls *wahr*, das Vorzeichen dieser Summe zu verwenden. Da das Vorzeichen genau das Spielerzeichen(`r`) ist, ist somit direkt der Gewinner bestimmt. Es müssen die Fälle Reihe, Spalte, Hauptdiagonale und Gegendiagonale wie bereits erwähnt separat behandelt werden, deshalb 4 Funktionen für eine Aufgabe…

Spur und Gegenspur In der Mathematik gibt es für die »Diagonalensumme« eine Verallgemeinerung, nämlich die Spur. Die Spur ist die Summe der Diagonalelemente, also die Summe jener Elemente, deren Indizes *gleich* sind. $\mathrm{Sp}(a) = \delta_{ij} a_{ij} = a_{11} + a_{22} + a_{33}$. [iv]
Die Gegenspur ist Summe der Gegendiagonalelemente, also die Summe jener Elemente, deren Summe der Indizes 4 ergibt. $\forall a_{ij} : i + j = 4$ Also $\mathrm{GSp}(a) = a_{13} + a_{22} + a_{31}$.
Diese Erkenntnisse wurden, so gut es ging, direkt in C-Code eingebaut.

1.4. Computergegner

Der Computer muss den momentanen Stand des Spielbrettes berücksichtigen, den menschlichen Spieler am Gewinnen hindern und möglichst schnell selbst Siegen. Diese Ziele werden erfüllt.
Die Vorgehensweise ist, jedem Spielbrettelement `board[i][j][0]` eine *Wertung* oder *Score* zuzuordnen. Diese Zuordnung wird von der `sync_coboard`-funktion gemacht, die Ergebnisse werden in `board[i][j][1]` gespeichert. (Siehe auch Abb. 1.4) Die Felder sind für den Sieg verschieden hilfreich:

1. a_{22}, das Feld in der Mitte, bietet 4 Siegmöglichkeiten
2. $a_{11}, a_{13}, a_{31}, a_{33}$, die Eckfelder, bieten jeweils 3 Möglichkeiten,

[iii] Die Zahl 61 steht laut ASCII für das Gleichheitszeichen »=«.

[iv] *Kronecker-Delta*, $\delta_{ij} = \begin{cases} 1 & \text{für } i = j \\ 0 & \text{für } i \neq j \end{cases}$

3. a_{12}, a_{21},a_{23},a_{32}, die restlichen Randfelder, bieten nur 2 Möglichkeiten.

Ist das Spielfeld leer, so wird der Computer die Felder in dieser Reihenfolge bevorzugen. Die Wertungsmatrix `board[i][j][1]` wird also wie folgt initialisiert:

$$\texttt{board[i][j][1]} = \left(b_{ij}\right) = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} 2 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 1 & 2 \end{pmatrix} \tag{1.2}$$

Nach jedem Zug wird das Wertungsbrett mit dem echten Spielbrett synchronisiert, (aktualisiert). Jedoch werden nur jene Felder aktualisiert, welche noch beschreibbar (leer) sind.

1.4.1. Feldbewertungs-Algorithmus

In `sync_coboard` werden Änderungen am Spielbrett ins Wertungsbrett übernommen, wobei jedes Feld einzeln bewertet wird. Zuerst werden bereits beschriebene Felder ausgeschlossen (Score auf 0 gesetzt).

Verhinderung des Gegnersieges Bei den Freien Feldern wird geprüft, ob deren Nachbarn (in Reihe, Spalte und Diagonale) gleich sind. Ist dies der Fall, bedeutet das, dass die R./S./D. fast voll ist! Setzt man auf das betrachtete Feld, hat dies einen Sieg zur Folge. Dem Algorithmus ist es egal, ob der Mensch oder Computer einen Zug vom Sieg entfernt ist, das Feld wird um 60 Punkte höher gewertet.

Summen Andernfalls (Summen ungleich ±2) wird einfach der Betrag der Summe im Realfeld(`board[i][j][0]`) dazugezählt. Das hat zur Folge, dass nichtleere R./S./D. leicht bevorzugt werden. (Im Gegensatz zu ganz Leeren.)

Gewichtung Die Gewichtung erfolgte nach Bauchgefühl und Beobachtungen bei mehreren Spieldurchgängen. Sie könnte bestimmt noch optimiert und ausgebaut werden, jedoch funktioniert sie in dieser Form »gut genug«. Es wurde beispielsweise beobachtet, dass der Computer bei Vorhandensein von 2 unmittelbaren Gewinnmöglichkeiten (für ihn), davor noch den Gegnersieg verhindert.
(Was nicht passieren sollte, der Computer sollte so schnell wie möglich gewinnen!) Jedoch ist es in diesem Fall nicht weiter schlimm, da der Computer ohnehin zwei Siegmöglichkeiten hatte. Ein solcher Fehler konnte bei einer Siegmöglichkeit (fatal) nicht reproduziert werden.

Anmerkung Es wird hier ein *deterministischer Algorithmus* also ein Algorithmus, welcher mit gleichen Startbedingungen stets die gleichen Ergebnisse liefert, »beobachtet«. So einen Algorithmus müsste man nicht beobachten, da alle Parameter Antworten auf alle möglichen Inputs bekannt sind. Man könnte ihn in mit mathematischen Methoden optimieren.

1.4.2. Zugkandidat

Der Kandidat des Computers wird ermittelt, wenn der Computer dran ist. (In Funktion `do_move`) Es wird über alle Elemente des Wertungsfeldes iteriert, und der höchste Score festgehalten. Das Maximum des Wertungsfeldes wird gesucht.

1.5. Hauptprogramm

Das Hauptprogramm ist zu lang, man könnte hier 8 Zeilen[v] in eine »Start-up«-funktion packen, aber dann müsste man sich wieder die Pointer umbauen, was ich nicht wollte.

Zähler Es werden die Zähler `r` und `cpn` (counter-player-name) initialisiert. Diese werden verwendet um die Spielerzahlen und Symbole sowie den Prompt und Spielernamen bei jedem Zug auszutauschen.

Hauptschleife Zuerst wird der Bildschirm geleert, dann wird das Wertungsbrett mit dem Spielbrett synchronisiert. Das Brett (oder die Bretter im Fall `-e -debug`) werden ausgegeben.
Nun wird geprüft, ob ein Sieg vorliegt und gegebenenfalls das Spiel verlassen.
Falls nicht, wird der Prompt ausgegeben und wenn der menschliche Spieler am Zug ist, auf Eingabe gewartet.
Es wird die Zugroutine aufgerufen.

Fehler - Aufwärtszählen nach ungültiger Eingabe Gibt der Spieler eine Ungültige Eingabe ein, so wird diese in der `do_move`-funktion abgefangen. Da `invert_player` nicht aufgerufen wird, bleibt der Spieler am Zug, jedoch wird unter anderem das Wertungsbrett aktualisiert, was zu einer Erhöhung aller Felder um ein paar Punkte führt, ein möglicher Überlauf nach entsprechenden Wiederholungen der ungültigen Eingabe könnte einen Overflow zur Folge haben. Ich weiß nicht wann der Buffer überläuft (nicht bei 255). Die Matrizen sehen nach gut 100 falschen Eingaben so aus (Gl. 1.3).

$$\begin{aligned} \texttt{board[i][j][1]} = \left(a_{ij}\right) &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad i,j \in \{1,2,3\} \\ \texttt{board[i][j][1]} = \left(b_{ij}\right) &= \begin{pmatrix} 3 & 158 & 315 \\ 158 & 0 & 314 \\ 315 & 314 & 0 \end{pmatrix}, \quad i,j \in \{1,2,3\} \end{aligned} \tag{1.3}$$

[v] 347 bis 354

```
+---+-------------------+   +---+---------------------------------+
| 1 | O O O O O X X X X |   | 1 |   0   0   0   0 0   0   0   0 0 |
| 2 | O X O X X O O O X |   | 2 |   0   0   0   0 0   0   0   0 0 |
| 3 |     O X O O O O X |   | 3 | 144 122   0   0 0   0   0   0 0 |
| 4 | O O O X X     O X |   | 4 |   0   0   0   0 0 128  89   0 0 |
| 5 |         O O     X |   | 5 |  97  75 114 122 0   0  76 102 0 |
| 6 | O   O O O O     X |   | 6 |   0 101   0   0 0   0 102 128 0 |
| 7 | X O X O O O O O O |   | 7 |   0   0   0   0 0   0   0   0 0 |
| 8 | X X X X X X X X X |   | 8 |   0   0   0   0 0   0   0   0 0 |
| 9 | X X X O X X X X X |   | 9 |   0   0   0   0 0   0   0   0 0 |
+---+-------------------+   +---+---------------------------------+
  X | 1 2 3 4 5 6 7 8 9 |     C |   1   2   3   4 5   6   7   8 9 |
    +-------------------+       +---------------------------------+
Computer hat mit X gewonnen!
```

Abbildung 1.5.: Spielverlauf 9 × 9-brett

```
+---+---+   +---+---+     +---+-----+   +---+-------+
| 1 | O |   | 1 | 0 |     | 1 | O   |   | 1 | 0 423 |
+---+---+   +---+---+     | 2 | X O |   | 2 | 0   0 |
  O | 1 |     C | 1 |     +---+-----+   +---+-------+
    +---+       +---+       O | 1 2 |     C | 1   2 |
Simon hat mit O gewonnen!     +-----+       +-------+
```

Abbildung 1.6.: Verschiedene Spielenden mit variierenden Brettgrößen

1.6. Erweiterung des Spielbrettes

Die Größe des Spielbrettes wird nun von 3 × 3 auf $n \times n$ vergrößert, ohne viel neuen Code entwerfen zu müssen. Es wurde ein globales Arraymaximum **m**ax **s**ize eingeführt, bislang noch mittels `#define ms 10` und jedes vorkommen von `board[4][4]` mit `board[m][ms]` ersetzt. Analog wurden die Gewinn-Feststellungsalgorithmen und die Computerstrategie auf `ms` generalisiert. Der aufwändigste Teil war die Verallgemeinerung der bislang noch weitgehend statischen Funktion `draw_board`. Damit ist es möglich, das Spiel bis zu einer Größe von 9 × 9 Feldern korrekt darzustellen (die Zahl 10 hat 2 Stellen und sprengt damit das Gitter). Die Algorithmen sind wahrscheinlich nur durch ihre (In)effizienz beschränkt.
Im Prinzip wird alles mit der Brettgröße skaliert das Prinzip bleibt gleich. Diese Erweiterung auf ein $n \times n$-Feld funktioniert nur wenn für einen Sieg immer noch n Felder vom gleichen Spieler besetzt sein müssen. Also die volle Länge, Breite oder Diagonale.

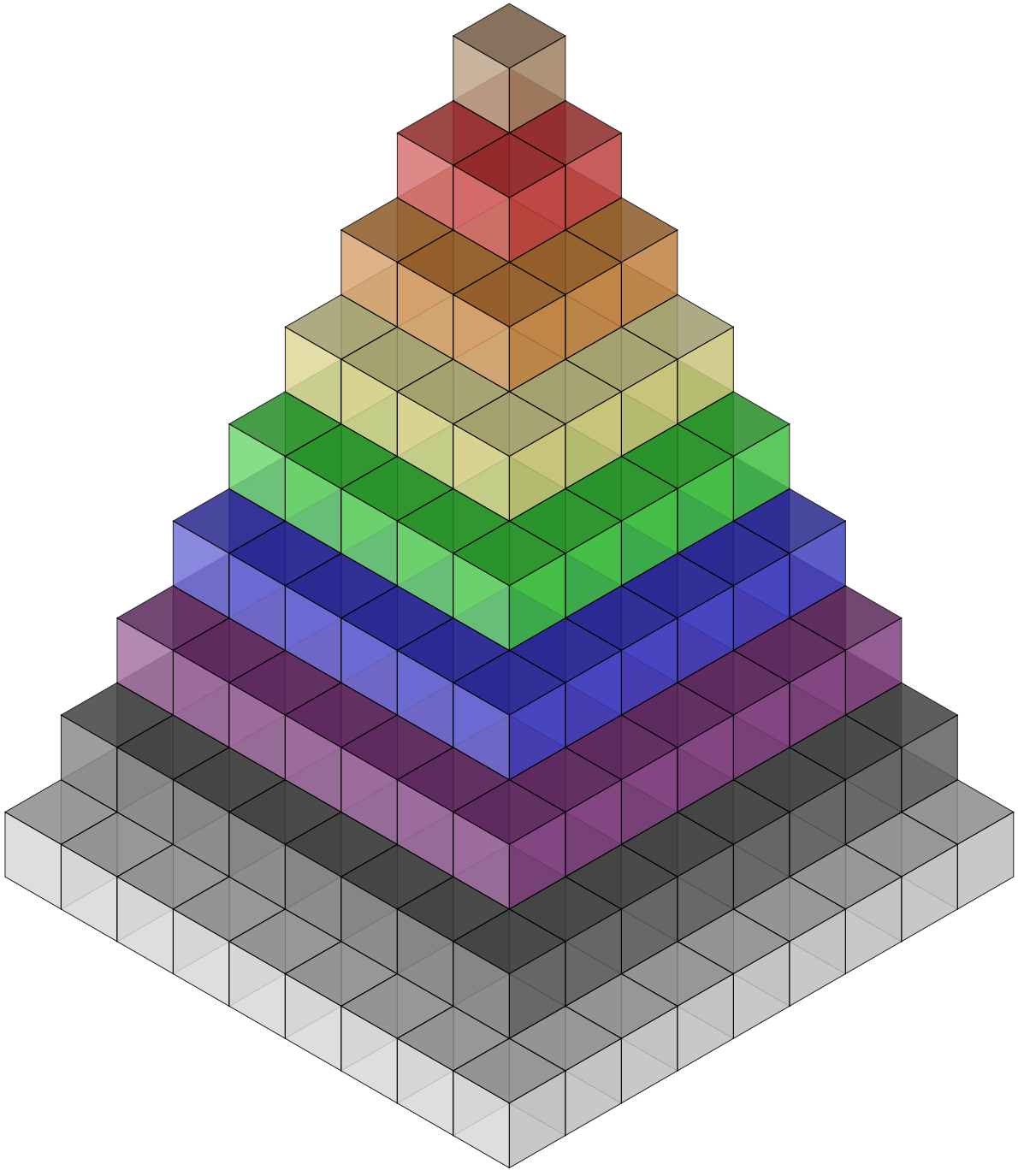

Abbildung 1.7.: Alle korrekt darstellbaren Spielbrettgrößen, von 1×1 bis 9×9.
Die Wertungsfelder und 0-reihen und 0- Spalten werden hier nicht gezeigt.

Anhang A.

Quellcode

A.1. Code: Initialisierungen

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Globales Arraymaximum
#define ms 6
// Deklaration des Spielbretts
int board[ms][ms][2];

// Flag für debugging - Information.
int extascii, debug = 0;

/*
   Betragsfunktion abs(x) kommt aus stdlib.h, siehe 'man 3 abs'
   Signum-Funktion sgn(x), keine Library gefunden, eigene Implementation.
*/
int sgn(int x)
{
  if(x == 0) return 0;
  if(x > 0) return 1;
  if(x < 0) return -1;
  else exit(127);
}

// Abbildungsfunktion für die Spielbrettausgabe.
// Diese Zeichen erscheinen auch in der linken unteren Ecke bei Spielende
char map_numtochar(int r)
{
```

```
29    if(r == 0) return ' ';
30    else if(r == 1) return 'X';
31    else if(r == -1) return 'O';
32    else if(r == '=') return '=';
33    else return r;
34  }
35  // Füllt ganze Matrix mit dem Inhalt der Variable i.
36  void init_board(int i,int board[ms][ms][2])
37  {
38    for(int j=0;j<ms;j++)
39    {
40      for(int k=0;k<ms;k++)
41      {
42        board[j][k][0] = i;
43        board[j][k][1] = i;
44        // Füllen der Wertungsmatrix.
45        if(j==k || (j+k)==ms) board[j][k][1]=3;
46        else board[j][k][1]=2;
47      }
48    }
49  }
```

A.2. Code: Summenfunktionen

```
50  // Summe jener Elemente board[x][y] mit gleichem x.
51  int col_sum(int x,int board[ms][ms][2])
52  {
53    int acc_c = 0; // Akkumulator Spaltensumme
54    for(int r=1;r<ms;r++)
55    {
56      // Über r (row) wird iteriert.
57      acc_c += board[r][x][0];
58    }
59    //board[0][x][0] = acc_c;
60    return acc_c;
61  }
62
63  // Summe jener Elemente board[x][y] mit gleichem y.
64  int row_sum(int y,int board[ms][ms][2])
65  {
66    int acc_r = 0; // Akkumulator Zeilensumme
67    for(int c=1;c<ms;c++)
```

```
68    {
69      // Über c (column) wird iteriert.
70      acc_r += board[y][c][0];
71    }
72    //board[y][0][0] = acc_r;
73    return acc_r;
74  }
75
76  // Die Spur der Matrix, Summe der Hauptdiagonalelemente
77  // Zu Beginn ist die Matrix Spurfrei, d.h. trace(board) = 0
78  int trace(int board[ms][ms][2])
79  {
80    int acc_t = 0; // Akkumulator Spur (trace)
81    for(int a=1;a<ms;a++)
82    {
83      acc_t += board[a][a][0];
84    }
85    //board[0][0][0] = acc_t;
86    return acc_t;
87  }
88  // Gegenspur, Summe der Gegendiagonalelemente
89  // Die Summe jener Elemente, deren Summe der Indizes 4 ergibt.
90  // ∀a ∈ ℝ : a = sgn(a) · abs(a)
91  int anti_trace(int board[ms][ms][2])
92  {
93    int acc_at = 0;
94    for(int r=0;r<ms;r++)
95    {
96      for(int c=0;c<ms;c++)
97      {
98        if((r+c)==(ms-1)) acc_at += board[r][c][0];
99      }
100   }
101   return acc_at;
102 }
```

A.3. Code: Siegprüfung

```
103 // Produkt aller Elemente, falls 0, dann gibt es noch leere Felder.
104 // Das Produkt kann nicht groß werden, da Elemente in
105 // {+1,0,-1} enthalten sein müssen.
106 int prod(int board[ms][ms][2])
```

```
107 {
108   int acc_p = 1; // Akkumulator Produkt
109   for(int j=1;j<ms;j++)
110   {
111     for(int k=1;k<ms;k++)
112     {
113       acc_p = (acc_p * board[j][k][0]);
114     }
115   }
116   return acc_p;
117 }
118 int wincheck()
119 {
120   // ∀a ∈ ℝ : a = sgn(a) · abs(a)
121   if(abs(trace(board)) == (ms-1)) return sgn(trace(board));
122   if(abs(anti_trace(board)) == (ms-1)) return sgn(anti_trace(board));
123 
124   for(int i=1;i<ms;i++)
125   {
126     if(abs(row_sum(i,board)) == (ms-1)) return sgn(row_sum(i,board));
127     if(abs(col_sum(i,board)) == (ms-1)) return sgn(col_sum(i,board));
128   }
129   // Die Spielbrettmatrix eines unfertigen Spiels enthält mindestens eine 0.
130   if(prod(board) == 0) return 0;
131   // Die Zahl 61 entspricht dem '='-Zeichen.
132   // Das '='-Zeichen wird in der print_board-Funktion direkt ausgegeben.
133   else return '=';
134 }
135 
136 int find_winner()
137 { // guards wären hier cool.
138   if(wincheck() == 0) return 0;
139   if(wincheck() == 1) return 1;
140   if(wincheck() == -1) return 2;
141   if(wincheck() == 61) return 3;
142   else exit(127);
143 }
144 
145 void exithandler(int p, char player_name[2][10])
146 {
147   if(p==0) return;
148   else if(abs(p)==1)
149   {
```

```
150     int np = ((p + 1) / 2);
151     printf("%s hat mit %c gewonnen!\n",player_name[np],map_numtochar(p));
152   }
153   else if(p==61)
154     printf("Unentschieden!");
155   exit(0);
156 }
```

A.4. Code: Spielbrettausgabe, diverses

```
157 // Gibt das Brett aus.
158 void draw_board(int board[ms][ms][2])
159 {
160   if(extascii==1 && debug==1)
161   {
162     for(int z=1;z<ms;z++)
163     {
164       if(z==1)
165       { // Zeile 1
166         printf("");
167         for(int i=0;i<2*ms;i++)
168         {
169           if(i<ms)printf("");
170           if(i==ms)printf("\b   ");
171           if(i>ms)printf("");
172         }
173         printf("\n");
174       } // Beginn Zeile 2
175       printf(" %d  ",z);
176       for(int s=1;s<ms;s++)
177         printf("%c ",map_numtochar(board[z][s][0]));
178       printf("    %1d  ",z);
179       for(int s=1;s<ms;s++)
180         printf("%d ",board[z][s][1]);
181       printf("\n");
182     }
183     printf("");
184      for(int i=1;i<(2*ms);i++)
185     {
186       if(i<ms) printf("");
187       if(i==ms) printf("   ");
188       if(i>ms&&i<(2*ms)) printf("");
```

```
189       }
190       printf("\n  %c  ",map_numtochar(wincheck()));
191       for(int i=1;i<(2*ms);i++)
192       {
193         if(i<ms) printf("%d ",i);
194         if(i==ms) printf("     C  ");
195         if(i>ms&&i<(2*ms)) printf("%d ",i-ms);
196       }
197       printf(" \n    ");
198       for(int i=0;i<2*ms;i++)
199           {
200             if(i<ms) printf("");
201             if(i==ms) printf("\b       ");
202             if(i>ms) printf("");
203           }
204           printf("\n");
205   }
206
207   else if(extascii==1 && debug==0)
208   {
209     for(int z=1;z<ms;z++)
210     {
211       if(z==1)
212       {
213         printf("");
214         for(int i=0;i<ms;i++) printf("");
215         printf("\b\n");
216       } // Beginn Zeile 2
217       printf(" %d  ",z);
218       for(int s=1;s<ms;s++)
219         printf("%c ",map_numtochar(board[z][s][0]));
220       printf("\n");
221     }
222     printf("");
223      for(int i=1;i<ms;i++) printf("");
224     printf("\n  %c  ",map_numtochar(wincheck()));
225     for(int i=1;i<ms;i++)
226       if(i<ms) printf("%d ",i);
227     printf(" \n    ");
228     for(int i=0;i<ms;i++) if(i<ms) printf("");
229         printf("\b\n");
230   }
231
```

```
232   else
233   {
234     for(int z=1;z<ms;z++)
235     {
236       if(z==1)
237       {
238         printf("+---+");
239         for(int i=0;i<2*ms;i++)
240         {
241           if(i<ms)printf("--");
242           if(i==ms)printf("\b+   +---+");
243           if(i>ms)printf("--");
244         }
245         printf("-+\n");
246       } // Beginn Zeile 2
247       printf("| %d | ",z);
248       for(int s=1;s<ms;s++)
249         printf("%c ",map_numtochar(board[z][s][0]));
250       printf("|   | %d | ",z);
251       for(int s=1;s<ms;s++)
252         printf("%d ",board[z][s][1]);
253       printf("|\n");
254     }
255     printf("+---+");
256      for(int i=1;i<(2*ms);i++)
257     {
258       if(i<ms) printf("--");
259       if(i==ms) printf("-+   +---+");
260       if(i>ms&&i<(2*ms)) printf("--");
261     }
262     printf("-+\n  %c | ",map_numtochar(wincheck()));
263     for(int i=1;i<(2*ms);i++)
264     {
265       if(i<ms) printf("%d ",i);
266       if(i==ms) printf("|     C | ");
267       if(i>ms&&i<(2*ms)) printf("%d ",i-ms);
268     }
269     printf("| \n    +");
270     for(int i=0;i<2*ms;i++)
271         {
272           if(i<ms) printf("--");
273           if(i==ms) printf("\b+       +");
274           if(i>ms) printf("--");
```

```
275             }
276             printf("-+\n");
277       }
278
279   }
280   // Gibt Hilfetext aus.
281   void show_help()
282   {
283       printf("Bitte geben Sie für die gewählte Stelle auf dem Spielfeld die entsprechend
284       printf("Ein Spieler hat gewonnen, wenn er es geschafft hat seine Zeichen so zu set
285       printf("dass er entweder 3 waagerecht, 3 senkrecht oder 3 diagonale in einer Reihe
286       printf("Der Prompt(X> oder O>) kennzeichnet Spieler 1 mit einem X und Spieler 2 mi
287       printf("Felder werden nach Zeile und Spalte bezeichnet, z.B. markiert O>23 die 2.Z
288       printf("Ist das Spiel zu Ende, findet man in der linken unteren Ecke des Brettes d
289       printf("Das Programm bietet ein alternatives User-Interface, indem man es mit 'jo4
290       getchar();
291       getchar();
292   }
293   // Leert Bildschirm
294   void clear_screen()
295   {
296     if(debug!=1) system("clear");
297   }
298   void invert_player(int** r, int** cpn)
299   {
300     **r = (0 - **r);
301     **cpn = ((**r + 1) / 2);
302   }
```

A.5. Code: Synchronisierung des Wertungsbretts

```
303   void sync_coboard(int board[ms][ms][2])
304   {
305     for(int l=1;l<ms;l++)
306     {
307       for(int m=1;m<ms;m++)
308       {
309         // Entferne im Realfeld schon besetzte Felder aus dem Wertungsfeld.
310         if(board[l][m][0]!=0) board[l][m][1]*=0;
311         // Wenn Feld noch leer, aktualisiere Feld, in Abhängigkeit seiner Nachbarn.
312         if(board[l][m][0]==0)
313         {
```

```
314         /*
315           Prüfe ob Zeilen/Spalten/Diagonalen fast voll sind also zwei gleiche Zeichen
316           (X,O)=Zahlen(+1,-1) drin sind.
317           Falls nicht, zähle den Betrag, von dem was drin ist dazu.
318           (Man kann da auch noch optimieren, also gewichten...)
319         */
320         if(abs(row_sum(l,board)) == (ms-2))
321         {
322           board[l][m][1] += (20*ms);
323           if(debug==1)
324           printf("Reihe %d ist fast voll, setze 20*Feldgröße drauf.\n",l);
325         }
326         else board[l][m][1] += abs(row_sum(l,board));
327
328         if(abs(col_sum(m,board)) == (ms-2))
329         {
330           board[l][m][1] += (20*ms);
331            if(debug==1)
332            printf("Spalte %d ist fast voll, setze 20*Feldgröße drauf.\n",m);
333         }
334         else board[l][m][1] += abs(col_sum(m,board));
335
336
337         // Wenn Das Feld auf der Hauptdiagonale liegt,
338         // berücksichtige diese Nachbarn auch.
339         if(l==m)
340         {
341           if(abs(trace(board)) == (ms-2))
342           {
343             board[l][m][1] += 20*ms;
344             if(debug==1)
345             printf("Hauptiagonale ist fast voll, setze 20*Feldgröße drauf.\n",l);
346           }
347           else board[l][m][1] += abs(trace(board));
348         }
349         // Bei allen Gegendiagonalelementen ist die Summe der Indizes = ms
350         // so kann man herausfiltern.
351         if(l+m==ms)
352         {
353           if(abs(anti_trace(board)) == (ms-2))
354           {
355             board[l][m][1] += (20*ms);
356             if(debug==1)
```

```
357               printf("Gegendiagonale ist fast voll, setze 20*Feldgröße drauf.\n",l);
358             }
359             else board[l][m][1] += abs(anti_trace(board));
360           }
361         }
362       }
363     }
364   }
```

A.6. Code: Zugroutine

```
365   void do_move(char answer[2],int* r,int* cpn,int board[ms][ms][2])
366   {
367     if(*r==1)
368     {
369       if(debug==1) printf("Der Computer ist dran.\n");
370
371       // Nächstes Feld, welches gesetzt wird
372       int x,y = 0;
373       int current_best = 0;
374       for(int a=1;a<ms;a++)
375       {
376         for(int b=1;b<ms;b++)
377         {
378           if(board[a][b][1] >= current_best)
379           {
380             current_best=board[a][b][1];
381             x=a;y=b;
382           }
383         }
384       }
385       if(debug==1) printf("Computer setzt auf %d%d weil Score dort %d ist.\n"
386       ,x,y,current_best);
387       // Prüfe zur Sicherheit, ob Feld wirklich leer,
388       // schreibe als Computer=X=+1 ins Feld.
389       if(board[x][y][0]==0)
390         {
391           board[x][y][0] = *r;
392           invert_player(&r,&cpn);
393         }
394     }
395     else
```

```
396   {
397     if(debug==1) printf("Der Spieler ist dran.\n");
398     if(answer[0]=='h') show_help();
399 
400     // Keine hinreichende Bedingung für Korrektheit der Eingabe, aber notwendig.
401     // Hier könnte man noch mehr erweitern (verscheidene Eingabestile parsen).
402     if(answer[0]!='\n' && answer[1]!='\n')
403     {
404       // Umwandeln von Char in Int
405       int x = (answer[0] - '0');
406       int y = (answer[1] - '0');
407       // Zulässige Eingaben (1,2,3)
408       if(x > 0 && y > 0 && x < ms && y < ms)
409       {
410         // Falls Feld leer(0) ist, setze Wert von r (dem Spieler) darauf(+1,-1)
411         if(board[x][y][0]==0)
412         {
413           board[x][y][0] = *r;
414           invert_player(&r,&cpn);
415         }
416         // Sonst, falls Feld bereits gleichen Wert enthält: Fehlermeldung
417         else if(board[x][y][0] == *r)
418         {
419           printf("F>Bereits von dir belegt!\n");
420 // Invertiere Rundenzahl (additiv).
421 // Einmalige Verwendung würde von der clear_screen Funktion überrollt werden.
422           getchar();
423           getchar();
424         }
425         // Falls Feld bereits von Gegner beschrieben (Feldinhalt ist = -r)
426         else if(board[x][y][0] == (0 - *r))
427         {
428           printf("F>Bereits vom Gegner belegt!\n");
429           getchar();
430           getchar();
431         }
432       }
433       // Unzulässige Eingabe
434       else
435       {
436         printf("F>Unzulässige Eingabe!\n");
437         getchar();
438         getchar();
```

```
439         }
440       }
441       else
442       {
443       printf("Bitte zwei Zahlen, Reihen und Spaltenzahl hintereinander ohne Leerzeichen
444       printf("Bestätigung des Zuges mit Enter. Beispiel: X>21[Enter]\n");
445       }
446     }
447   }
```

A.7. Code: Hauptprogramm

```
448   int main(int argc, char *argv[])
449   {
450     if(argc>1)
451     {
452       // Argumente werden ausgewertet für zusätzliche Funktionalitäten des Programmes.
453       for(int cnt=1;cnt<argc;cnt++)
454       {
455         if(strcmp(argv[cnt],"--debug")==0) debug=1;
456         if(strcmp(argv[cnt],"--extascii")==0) extascii=1;
457         else if(strcmp(argv[cnt],"-e")==0) extascii=1;
458       }
459     }
460     int cpn = 0;
461     // r=Rundenzähler, Abbildung von cpn={0,1} auf {-1,+1},
462     // repräsentiert momentanten Spieler (wie r).
463     int r = ((cpn * 2) - 1);
464     char answer[2];
465     char player_name[2][10] = {"Spieler","Computer"};
466     init_board(0,board);
467     printf("Willkommen zum TicTacToe gegen den Computer !\n");
468     printf("Wie heißt du ?\n");
469     scanf("%s",player_name[0]);
470     printf("Spieler %c, %s beginnt.\n",map_numtochar(r),player_name[cpn]);
471     printf("Für Hilfe geben Sie bitte 'h' ein und betätigen die Entertaste.\n");
472     printf("Zum Start des Spiels ENTER drücken.\n");
473     if(debug==1) printf("Debugmodus ist ein, Sie sehen mehr als nötig.\n");
474     getchar();
475     // Beginn der Spielzug-Schleife
476     while(1)
477     {
```

```
478     clear_screen();
479     // Synchronisiere Brett der möglichen Züge
480     sync_coboard(board);
481     draw_board(board);
482     exithandler(wincheck(),player_name);
483     if(debug==1) printf("%c(%s), r=%d, cpn=%d >"
484     ,map_numtochar(r),player_name[cpn],r,cpn);
485     else printf("%c(%s)>",map_numtochar(r),player_name[cpn]);
486     // Eingabe nur wenn der menschliche Spieler dran ist.
487     if(r==-1) scanf("%3s",answer);
488     else putchar('\n');
489     do_move(answer,&r,&cpn,board);
490   }
491 }
```

A.8. Rudimentärer Test mit dem Pipe-Operator

Man kann dem Programm eine Abfolge von Zügen im voraus mitgeben:

```
echo -e "\n23\n11\n31\n32\n12" | ./jo5 --debug
```