Visualisierung: Game of Life

Mahsa Mehrban
Markus Radtke

17. July 2020

1 Einleitung

Das Ziel dieses Projektes, im Rahmen der Vorlesung BM: Visualisierung, war es Conway's Game of Life [Gardener(1970)] zu implementieren. Es sollten editierbare und zufällige Startkonfigurationen ermöglicht werden. Weiterhin sollten alternative Regeln definiert und untersucht werden. Um dem Namen der Veranstaltung gerecht zu werden wurde ein Liveplot des Spiels implementiert, so wie eine Möglichkeit den Verlauf eines Spiels als Videodatei zu speichern.

2 Conway's Game of Life

Conway's Game of Life soll eine Anlehnung an den Aufstieg, Fall und die Variationsreichhaltigkeit der Gesellschaft von Organismen sein. Das Spiel erinnert mehr an eine Simulation, als an ein klassisches Spiel mit dem ein Mensch interagieren kann, und kann deshalb dem Simulationsspielgenre zugeordnet werden.

2.1 Die Regeln

Das Spiel wird auf einer Art Go-Brett (gemeint ist das chinesische Spiel *Go*) gespielt. Es ähnelt einer leeren, quadratischen Tabelle. Die einzelnen Quadrate auf dem Spielfeld symbolisieren die Zellen. Diese Zellen sollen an die bereits erwähnten biologischen Zellen erinnern, sie können den Zustand *lebendig* oder *tot* haben.

Bevor das Spiel beginnt muss eine Startkonfiguration erstellt werden. Dabei kann der sich Benutzer entweder selbstgewählte lebendige Zellen auswählen oder sich eine Konfiguration zufällig erzeugen. Conway wollte bestimmte Voraussetzungen für Startkonfigurationen erwirken.

1. Es sollte keine Startkonfiguration existieren, für die ein einfacher Beweis gefunden werden kann, dass die Bevölkerung grenzenlos wachsen kann.

2. Es sollten Startkonfigurationen existieren, die den Anschein erwecken, grenzenlos zu wachsen.

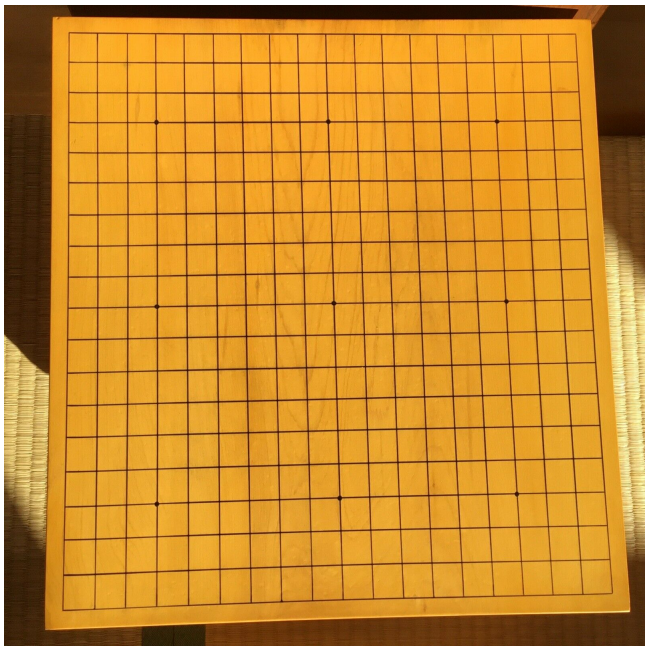

Abbildung 1: Foto eines Go-Bretts von oben.

3. Es sollten einfache Startkonfigurationen existieren, die wachsen und ihre Form ändern, bevor sie in einen der folgenden Zustände übergehen:
   (a) Komplettes Dahinschwinden: es gibt keine lebendigen Zellen mehr.
   (b) Stabiler Zustand: Die Anzahl der Zellen ändert sich nicht mehr und es oszillieren keine Zellen über die restlichen Generationen hinweg.
   (c) Oszillierender Zustand: Die Zellen wiederholen bestimmte Muster, welche aber niemals verschwinden oder aufhören. Die Wiederholung der Muster kann über mehrere Generationen erfolgen.

Jede Zelle hat acht Nachbarn auf dem Feld, vier davon liegen diagonal, die anderen vier orthogonal. Nach diesen Kriterien wurden die folgenden simplen Regeln für das Spiel erstellt. Die Regeln lauten wie folgt:

1. Überleben: Jede lebendige Zelle, welche genau zwei oder drei lebendige Nachbarn hat, überlebt die nächste Generation.
2. Sterben: Jede lebendige Zelle, mit weniger als zwei oder mehr als drei lebendigen Nachbarn stirbt (an Isolation/Überpopulation).
3. Geburten: Jede tote Zelle wird in der nächsten Generation zu einer lebendigen Zelle (also "geboren"), wenn sie genau drei lebendige Zellen als Nachbarn hat.

Die Startkonfiguration ist die erste Generation. Ein einmaliges Anwenden aller Regeln erzeugt die nächste Generation. Das heißt implizit, dass die Regeln sich nicht gegenseitig beeinflussen. Also beeinflussen die Geburten der dritten Regel nicht die ersten beiden Regeln und umgekehrt.

3 Technische Umsetzung

3.1 Programmiersprache und Packages

Die verwendete Programmiersprache in diesem Projekt ist Python in der Version 3.8. Wenn dieses Programm korrekt ausgeführt werden soll, muss auf die richtige Version geachtet werden. Die Mindestvoraussetzung sollte Python 3.6 sein, da allerdings unter 3.8 entwickelt wurde, ist dies allerdings sicherste Variante.

Als hilfreiche Python-Packages haben sich *numpy* und *matplotlib* erwiesen. *numpy* sollte aus der wissenschaftlichen Arbeit mit selbst geschriebenen Programmen bekannt sein, da es ein einfacher Weg ist komplexe Methoden auf viele verschiedene Datenstrukturen auszuführen. In unserem Projekt wird das „Spielbrett“ in einem *numpy* Array verwaltet. Unter anderem wird mit einer einfachen Methode berechnet, wie viele Zellen am Leben sind.

```
1  def how_many_live():
2      global board
3      wholesum = board.sum()
4      return wholesum
```

Listing 1: The how_many_live method

In 1 sieht man eine mächtige Anwendung von *numpy*. *board* ist das Array, in dem das aktuelle Spielfeld gespeichert ist. Es ist gefüllt mit 1en (für die lebendigen Zellen) und 0en (die toten Zellen). Mit der sum() Methode werden alle einzelnen Werte im board Array aufsummiert, welches die lebendigen Zellen in der aktuellen Generation ausgibt.

```
1  def init_board(size):
2      global board
3      board = np.zeros((size, size))
```

Listing 2: The init_board method

In 2 sieht ist die generische Initialisierung des *boards* zu sehen. Die Variable *board* ist global definiert und wird in Zeile 3 mit einem quadratischen Array der Größe *size* initialisiert. In dieser Methode wird noch keine Startkonfiguration angelegt, dies erfolgt in einem späteren Schritt.

matplotlib wurde für die grafische Darstellung des Spiels verwendet. Es ist einfach damit *numpy* Arrays als ein Bild darzustellen und anzuzeigen oder abzuspeichern. Für den Liveplot des Spiels war *matplotlib* essentiell, mit einer selbst geschriebenen Animationsfunktion kann das Spiel in Echtzeit angesehen werden. Dabei kann dabei die fps (frames per second), sowie die Anzahl der

Iterationen (Generationen) die man berechnen möchte, eingestellt werden. Dies erspart einem explizit definierte for-loops.

3.2 Programmstruktur

Das Zustand des Programms beim Zeitpunkt der Abgabe ist interaktiv. Der Benutzer startet das Programm mit *python game_of_life.py* und es folgt eine Reihe von Fragen, welche am Ende zur Simulation des Spiels führt.

```
1 if __name__ == '__main__':
2     rules = input('What ruleset do you want to use? (std/
          mone/pone)\n')
3     mode = input('Do you want graphic, generated or text
          file input? (gra/gen/txt):\n')
```

Listing 3: The main method

In 3 sind die ersten beiden Fragen des Programms zu lesen. In Zeile 2 wird der Benutzer gefragt nach welchen Regeln das Spiel gespielt werden soll. Dabei gibt es die Möglichkeiten:

- **'std'**: Es wird nach den normalen Regeln gespielt, wie sie von Conway aufgestellt wurden.
- **'pone'**: *pone* steht für „plus one“. Die Zahlenwerte der ursprünglichen Regeln werden um 1 erhöht.
- **'mone'**: *pone* steht für „minus one“. Die Zahlenwerte der ursprünglichen Regeln werden um 1 gesenkt.

Anschließend wird der Benutzer nach dem Inputmodus gefragt. Mit dem Input ist die Startkonfiguration des Spiels gemeint. Es gibt drei verschiedene Arten den Input als Nutzer zu gestalten.

- **'gra'**: steht für *graphic*. Wenn der Nutzer diese Option wählt, kann er später die einzelnen Zellen, die am Anfang leben sollen, mit der Maus in einem Array angeklickt werden. Dies ist praktisch für kleine Felder, wo evtl. bestimmte Muster testen möchte. Für größere Eingaben ist es eher nicht geeignet, weil es z. B. sehr lange dauert ein 100 x 100 Array ausreichend zu füllen und das GUI-Fenster nach einigen Eingaben, codebedingt, unresponsiv wird.
- **'gen'**: steht für *generated*. Wenn der Nutzer diese Option wählt, werden die lebendigen Zellen zufällig gesetzt. Es besteht die Möglichkeit dies später zu spezifizieren.
- **'txt'**: steht für Input durch eine .txt-Datei. In dieser Datei dürfen nur Nullen und Einsen mit einer Leertaste oder einem Zeilenumbruch getrennt stehen.

Im Weiteren wird die *main()* Methode durchlaufen, um die Fallunterscheidungen darzustellen.

Wenn die *generated* Option für den Input gewählt wurde, gelangt der Nutzer in einen if-case (gekürzt für die Anschaulichkeit):

```
1 percentage = input('The grid cells will be randomly
    chosen to be alive or dead.\nHow many cells should be
    alive (in percent)?\n')
2 board_size = input('The grid will be quadratic, what
    should be the size of a side?\n')
3 start_board = init_random_board(percentage, board_size)
4 how_many_live()
```

Listing 4: The main method

In 4 in Zeile 1 wird nach der Anzahl, in Prozent, der lebendigen Zellen gefragt und in der Variable *percentage* gespeichert. Dies bezieht sich auf die Startkonfiguration. Die Zahl kann zwischen 0 und 100 liegen. Gleitkommazahlen sind erlaubt. In Zeile 2 wird der Benutzer nach der Größe des Spielfeldes gefragt und in der Variable *board_size*, welche immer quadratisch ist. Dies wäre einfach zu ändern, jedoch ist dies bei uns fest verankert. Dies könnte auf Anfrage gemacht werden. In Zeile 3 wird dann das initiale Spielbrett erstellt, mit den zuvor abgefragten Parametern. Zur Überprüfung der Korrektheit der Anzahl der zufällig gewählten lebendigen Zellen wird die vorhin betrachtete Methode *how_many_live()* aufgerufen.

Wenn die *graphic* Option für den Input gewählt wurde, gelangt der Nutzer in diesen if-case (gekürzt für die Anschaulichkeit):

```
1 board_size = input('The grid will be quadratic, what
    should be the size of a side?\n')
2 print('End input with the enter key.')
3 grid = user_input_configuration(board_size)
4 board = grid
```

Listing 5: The main method

In Zeile 1, wird bereits wie im if-case zuvor, zuerst nach der *board_size* gefragt und in der Variable *board_size* gespeichert, wie bereits im if-case davor. In Zeile 2 ist ein wichtiger Hinweis für den Benutzer. In Zeile 3 folgt die grafische Eingabe der Zellen. Der Nutzer wird darauf hingewiesen, dass die Eingabe mit Enter beendet werden kann. Falls er das „X“ zum schließen des Fensters benutzt, wird das Programm abgebrochen. Die Erklärung des grafischen Inputs wird wegen Komplexität hier ausgelassen. Die wichtigste Funktion in der aufgerufenen Methode ist *ginput()* des *matplotlib* Packages.

Wenn die *txt* Option für den Input gewählt wurde, gelangt der Nutzer in die folgende Fallunterscheidung (gekürzt für die Anschaulichkeit):

```
1 filename = input('Name of your txt file: ')
2 board = txt_input(filename)
```

```
3  board_size = board.shape[0]
```

Listing 6: The main method

In Zeile 1 in 6 wird nach dem Dateinamen der Inputdatei gefragt. Dies kann auch ein absoluter Dateipfad sein. Anschließend wird in Zeile 2 der Inhalt der Datei in das *numpy* Array *board* gespeichert. Die *board_size* wird aus der Inputdatei abgeleitet.

Abschließend werden die jeweiligen Animationsmethoden für die jeweiligen Regelsätze gestartet.

```
1  if rules == 'std':
2      animate_std(iterations, interval)
3  elif rules == 'mone':
4      animate_m_one(iterations, interval)
5  elif rules == 'pone':
6      animate_p_one(iterations, interval)
```

Listing 7: The main method

Da diese gleich aufgebaut sind und sie sich nur in den Regeln unterscheiden, wird nur die Methode *animate_std()* erklärt. Die Argumente *iterations* und *interval* sind global vom Programm gesetzt und können nicht von Benutzer geändert werden, ohne den Code anzufassen.

```
1  def animate_std(frames, interval):
2      global global_fig
3      return FuncAnimation(global_fig, create_frame_std,
           frames=frames, interval=interval, blit=True,
           save_count=frames)
```

Listing 8: The main method

Diese Methode 8 verwendet die globale Variable *global_fig*, in welcher der Liveplot gespeichert wird. Wiedergegeben wird die *FuncAnimation()*, mit dem Parameter*global_fig*, um auf diese *figure* zu zeichnen. *create_frame_std()* ist die Methode, welche die Regeln anwendet und den Plot der neuen Generation wiedergibt. *frames* ist die Anzahl der Iterationen, also der Generationen von Zellen. *interval* ist die Latenz bevor ein neuer Frame (in Millisekunden) angezeigt wird. *blit* ist eine Option um die Animation flüssiger zu machen und das alte *board* aus der Animation zu entfernen. *save_count* ist eine Hilfe für das Speichern des Plots als Video.

Abschließend wird noch die Implementation der Regeln betrachtet. Um es kurz zu halten wird wieder nur eine Methode betrachtet, die der Standardregeln. Wichtig hierbei zu beachten ist, dass das Array so behandelt wird, als seien die Ränder feste Grenzen. Manche Implementationen benutzen toroidale Oberflächen, in diesem Projekt beschränken wir uns auf das Feld mit festen Grenzen. Die Zellen sterben, wenn sie die Grenze „überschreiten“ wollen. Deswegen sieht die Implementation der Regeln unübersichtlich aus, da die Sonderfälle am Rand behandelt werden mussten.

```
1  def apply_standard_rules():
2      global board
3      next_gen_board = np.zeros((board.shape[0], board.
           shape[1]), int)
4      for i in range(board.shape[0]):
5          for j in range(board.shape[1]):
6              tmp_sum = 0
7              if i == 0:
8                  t = 0
9              else:
10                 t = board[i - 1][j]
11             if i == 0 or j == board.shape[1] - 1:
12                 tr = 0
13             else:
14                 tr = board[i - 1][j + 1]
15             if j == board.shape[1] - 1:
16                 r = 0
17             else:
18                 r = board[i][j + 1]
19             if i == board.shape[0] - 1 or j == board.
                   shape[1] - 1:
20                 br = 0
21             else:
22                 br = board[i + 1][j + 1]
23             if i == board.shape[0] - 1:
24                 b = 0
25             else:
26                 b = board[i + 1][j]
27             if i == board.shape[0] - 1 or j == 0:
28                 bl = 0
29             else:
30                 bl = board[i + 1][j - 1]
31             if j == 0:
32                 l = 0
33             else:
34                 l = board[i][j - 1]
35             if i == 0 or j == 0:
36                 tl = 0
37             else:
38                 tl = board[i - 1][j - 1]
39             tmp_sum = t + tr + r + br + b + bl + l + tl
```

Listing 9: The apply_standard_rules method

Da die Regeln nicht in-place, also im selben Array wie die Ausgangsgeneration, ausgeführt werden können, wird zunächst in Zeile 3 ein Array *next_gen_board* mit der selben Größe des Ausgangsarrays erstellt. In den for-loops wird über

die Indizes der shapes des Arrays iteriert. i iteriert über die Zeilen des Arrays, j über die einzelnen Elemente in der Zeile. Für das aktuelle Element des Arrays *board*, also *board[i][j]* wird in den vielen else-if-Abfragen die Summe der Nachbarschaft gebildet. Diese Summe wird in der Variable *tmp_ sum* gespeichert, um anschließend die Regeln durchzusetzen.

```
1                # rules and transfer to next generation
2                # rule 1 survival
3                if tmp_sum == 2 or tmp_sum == 3 and board[i][
                     j] == 1:
4                    next_gen_board[i][j] = board[i][j]
5
6                # rule 2 deaths
7                elif (tmp_sum > 3 or tmp_sum < 2) and board[i
                     ][j] == 1:
8                    next_gen_board[i][j] = 0
9
10               # rule 3 births
11               elif tmp_sum == 3 and board[i][j] == 0:
12                   next_gen_board[i][j] = 1
13
14               else:
15                   next_gen_board[i][j] = board[i][j]
16       board = next_gen_board
```

Listing 10: The apply_standard_rules method

Regel 1 schaut, ob *tmp_ sum* gleich zwei oder drei ist und ob das aktuell betrachtete Element eine eins, also am Leben, ist. Wenn dies zutrifft, wird das Feld im zweiten Array *next_ gen_ board* auf eins gesetzt. Regel 2 betrachtet die Zellen, die sterben werden. Wenn die Nachbarschaftssume kleiner als zwei oder größer als drei ist, wird das aktuelle Element in der nächsten Generation auf null gesetzt. Die dritte Regel betrachtet nur die toten Zellen und belebt sie in der nächsten Generation, wenn die Nachbarschaftssumme genau gleich drei ist. Im letzten else, werden die restlichen Zellen, die nicht verändert worden sind, noch übertragen. Zum Schluss wird die globale Variable *board* auf die Werte des *next_ gen_ board* gesetzt, um für die nächste Anwendung der Regeln bereit zu stehen.

4 Auswertung

Wie in der vorigen Kapiteln bereits erwähnt, wurden drei verschiedene Regelwerke betrachtet.

1. Das normale Regelwerk *(standard)*
2. Das normale Regelwerk, aber alle Zahlen mit 1 subtrahiert *(m_ one)*

3. Das normale Regelwerk, aber alle Zahlen mit 1 addiert *(p_ one)*

Jede dieser Regeln wurde mit der *generate* Funktion simuliert. Zur Erinnerung: Die *generate* Funktion hat eine Prozentzahl der gewünschten lebendigen Zellen als Argument entgegen genommen und diese zufällig in der Startkonfiguration gesetzt. Als Parameter für die Anzahl der Generationen wurde 500 gewählt. Es wurde so gewählt, da es eine hohe Anzahl an Iterationen darstellt und somit schnelle Tode einer Zellengesellschaft aufzeigen kann. Andererseits ist es lang genug, um auch stabile Zustände darzustellen. Wahrscheinlich ist diese Zahl für manche Zustände, die sich nach der 500. Generation auflösen, zu niedrig. Es ist jedoch eine gute Balance zwischen Rechenzeit und Ergebnis. Um einen Überblick über den Verlauf der Spiele zu bekommen, wurden Wahrscheinlichkeiten der lebendigen Zellen von 0 bis 100 gewählt, wobei in Fünferschritten inkrementiert wurde. Für die Analyse wurde der Start- und Endzustand als Bild gespeichert und hier betrachtet. Weil der Umfang zu groß wäre, wenn alle Bilder betrachtet werden würden, wird nur eine Auswahl gezeigt, bei denen der Unterschied ersichtlich wird.

4.1 standard

Eine unwichtige Betrachtung ist die mit Wahrscheinlichkeit p = 0.00. Alle Felder sind am Anfang tot und könne mit einem schnellen Blick auf Regel 3 auch nicht zum Leben erweckt werden. Ebenso verhält es sich mit p = 1. In der zweiten Generation sind nur vier Felder übrig, in der dritten Generation existiert keine lebendige Zelle mehr. Interessant zu sehen ist, dass bereits bei p = 0.05 ein stabiler oder oszillierender Zustand zu erreichen ist. Im Startzustand leben 534 Zellen, im letzten Zustand leben 63. Es ist eine starke Reduktion an lebendigen Zellen zu erkennen, jedoch erfolgt kein kompletter Tod der Gesellschaft. Es sind auch bestimmte Formen zu erkennen, die entweder den stabilen oder oszillierenden Zustand erreicht haben.

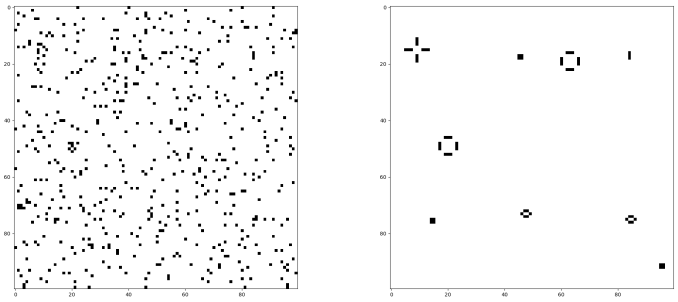

Abbildung 2: l. erster Frame, r. letzter Frame. p=0.05, 100x100 Array

Tabelle 1: p = Wahrscheinlichkeit lebendiger Zellen, start = lebendige Zellen im ersten Frame, last = lebendige Zellen im letzten Frame, label=tbl:std

p	start	last
0.00	0	0
0.05	100	63
0.10	963	364
0.15	1397	345
0.20	2007	298
0.25	2496	397
0.30	2937	391
0.35	3418	283
0.40	4019	481
0.45	4619	548
0.50	5069	338
0.55	5611	595
0.60	6039	429
0.65	6440	471
0.70	6939	344
0.75	7551	85
0.80	8072	24
0.85	8468	11
0.90	8983	4
0.95	9520	4
1	10000	0

In Tabelle 1 ist zu sehen, dass die lebendigen Zellen im ersten Frame der Konfiguration entsprechen. Was allerdings auffällt: Die Anzahl der lebendigen Zellen im letzten Frame von den Wahrscheinlichkeiten p = 0.1 bis p = 0.7 sind weitestgehend ähnlich. Alle sind im Bereich 300 - 500 lebendiger Zellen. Wenn die Simulationen erneut durchgeführt werden würde, könnte wahrscheinlich noch mehr Variation entdeckt werden. Dies hat mit der zufälligen Initialisierung der Zellen und deren Position zueinander zu tun. Was allerdings eindeutig erkennbar ist, dass die Anzahl der überlebenden Zellen drastisch sinkt je stärker sich p ab p = 0.75 in Richtung 1 bewegt. Bei p = 0.9 und p = 0.95 sind nur vier Zellen übrig geblieben. Dies hätten je nach Simulation auch null sein können.

4.2 m_one

Die Beobachtung von den Standardregeln lässt sich bei p = 0.00 und p = 1 auch beobachten. Interessant ist dazu noch, dass bei p = 0.9 und p = 0.95 ebenfalls keine Zellen überleben. Ähnlich wie bei den Standardregeln, fällt sofort auf, dass sich die Zahlen der lebendigen Zellen im letzten Frame in einem Bereich aufhalten. Von p = 0.05 bis p = 0.85 ist dieser Bereich bei 2800 - 3000 und

ändert sich danach drastisch zu der bereits angesprochenen null. Weiterhin fällt auf, dass die Population gegenüber dem ersten Frame deutlich gestiegen ist. Da die Regeln zum Überleben deutlich lascher sind, ergibt dies Sinn. Eine komplette Überpopulation wird durch die etwas laschere, aber trotzdem greifende Regel 2 verhindert.

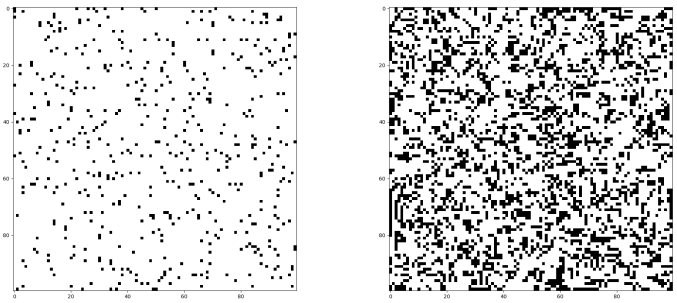

Abbildung 3: l. erster Frame, r. letzter Frame. p=0.05, 100x100 Array

Im Vergleich zu 2 ist der letzte Frame in 3 deutlicher mit Zellen besiedelt. Dies ist auf die lascheren Überlebensregeln zurückzuführen.

Tabelle 2: p = Wahrscheinlichkeit lebendiger Zellen, start = lebendige Zellen im ersten Frame, last = lebendige Zellen im letzten Frame, label=tbl:mone

p	start	last
0.00	0	0
0.05	502	3000
0.10	995	2901
0.15	1547	2823
0.20	1989	2843
0.25	2472	2842
0.30	2973	2855
0.35	3478	2977
0.40	3944	2940
0.45	4436	3073
0.50	4975	2929
0.55	5531	2893
0.60	6002	2941
0.65	6593	2913
0.70	7010	3001
0.75	7469	3044
0.80	7989	2965
0.85	8567	2984
0.90	8954	0
0.95	9556	0
1	10000	0

4.3 p_one

Auch bei diesem Regelwerk können die selben Beobachtungen gemacht werden. Bei den Extremwerten p = 0.00 und p = 1 gibt es keine überlebenden Zellen im letzten Frame. Ähnlich wie bei den *m_one* Regeln sind auch Werte für die lebendigen Zellen im letzten Frame, die p = 1 entgegen gehen, null. Allerdings beginnt dies bereits eine Wahrscheinlichkeitsstufe früher, nämlich bei p = 0.85. Generell ist zu beobachten, dass die Anzahl der überlebenden Zellen sehr gering ist. Dies ist allerdings auch zu erwarten, da die Anforderungen zum Überleben, die die Regelbildung impliziert, deutlich härter für die Zellen sind.

In 4 zeigt sich, was erste Überlegungen bestätigen. Da die Überlebenskriterien für die Zellen in diesem Regelwerk härter sind, existieren in diesem Beispiel am Ende gar keine Zellen mehr.

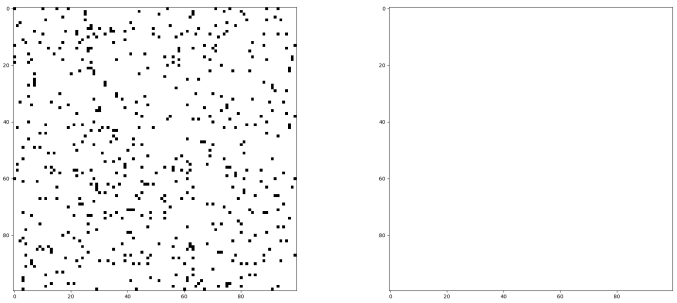

Abbildung 4: l. erster Frame, r. letzter Frame. p=0.05, 100x100 Array

Tabelle 3: p = Wahrscheinlichkeit lebendiger Zellen, start = lebendige Zellen im ersten Frame, last = lebendige Zellen im letzten Frame, label=tbl:mone

p	start	last
0.00	0	0
0.05	497	0
0.10	1033	8
0.15	1495	16
0.20	1994	37
0.25	2517	84
0.30	3039	72
0.35	3562	108
0.40	3974	120
0.45	4462	80
0.50	4957	73
0.55	5506	123
0.60	6036	92
0.65	6417	96
0.70	6997	48
0.75	7502	20
0.80	7961	16
0.85	8497	0
0.90	8960	0
0.95	9512	0
1	10000	0

5 Unbetrachtetes

Dieses Kapitel handelt von den Regeln oder Umgebungen, die in diesem Projekt nicht beachtet wurden, jedoch interessant sein könnten, wenn ein größerer

Arbeitsaufwand gerechtfertigt werden kann.

5.1 Änderung von betrachteten Nachbarn

Eine Möglichkeit, die analysiert werden könnte, wäre die Änderung der Anzahl an Nachbarn. Gemeint ist damit, dass bei einem Go-Brett nicht alle acht Nachbarn, sondern z. B. nur vier Nachbarn betrachtet werden. Entweder nur die diagonalen oder die orthogonalen oder vielleicht sogar zufällig gewählte, aber feste in ein Spiel.

Was sofort einleuchtet ist, dass die Überlebenswahrscheinlichkeiten der Zellen deutlich abnimmt, wenn die Zahlenwerte der Regeln nicht angepasst werden. Ähnlich wie es bei dem Regelwerk *p_ one* passiert ist.

Es müsste also eine Anpassung der Regeln erfolgen. Ein Schritt, der sofort einfällt, ist es wie bei *m_ one* alle Zahlen zu reduzieren. Dies ist jedoch nur beschränkt möglich, wenn der Tod durch Isolation nicht ignoriert werden soll.

5.2 Änderung der Struktur des Spielfeldes

Eine weitere Möglichkeit andere Ergebnisse zu erzielen ist es, das Spielfeld nicht in gleichmäßige Quadrate zu unterteilen, sondern z. B. in Hexagone oder Dreiecke oder ähnliche Strukturen.

Wenn die Hexagone als Beispiel genommen werden, dann fällt auf, dass diese jeweils nur sechs Nachbarn haben. Wenn die Regeln also nicht geändert werden, klingt es plausibel, dass die Zellen eine höhere Wahrscheinlichkeit zum Überleben haben. Allerdings ist es auch nicht ausgeschlossen, dass dadurch öfter der Fall der Überpopulation auftritt und Zellen wieder sterben können.

Literatur

[Gardener(1970)] M Gardener. Mathematical games: the fantastic combinations of john conway's new solitaire game "life. 1970.