

# Das Lösen und Generieren von Sudokus

Lisa Rüther und Rick Simon

28. März 2018

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>4</b>
<b>2 Vorhaben/Ziele</b>	<b>4</b>
2.1 Allgemein . . . . .	4
2.2 Sudoku-Lösungsprogramm . . . . .	4
2.3 Sudoku-Generator . . . . .	4
<b>3 Sudoku-Lösungsprogramm</b>	<b>5</b>
3.1 Entstehung . . . . .	5
3.1.1 Leeres Sudoku . . . . .	5
3.1.2 Einträge im Sudoku verändern . . . . .	7
3.1.3 Ausschluss-Prinzip für ein Feld . . . . .	8
3.1.4 Ausschluss-Prinzip für ein Kästchen . . . . .	9
3.1.5 Ausschluss-Prinzip für Zeilen und Spalten . . . . .	10
3.1.6 Double-Ausschluss . . . . .	10
3.1.7 Naked-Subset . . . . .	12
3.1.8 Force-Chain . . . . .	14
3.2 Gelöste Rätsel . . . . .	18
<b>4 Sudoku-Generator</b>	<b>19</b>
4.1 Generierungsprozesse und Experimente . . . . .	19
4.1.1 Generierung vollständiger Sudokus . . . . .	19
4.1.2 Test 1: Sudoku generieren durch zufälliges herausnehmen von Zahlen . . . . .	21
4.1.3 Test 2: Sudoku generieren durch zufälliges Hinzufügen ohne Regel . . . . .	25
4.1.4 Test 3: Sudoku generieren durch zufälliges Hinzufügen mit mäßigen Regel . . . . .	25
4.1.5 Test 4: Sudoku generieren durch zufälliges Hinzufügen mit strengen Regel . . . . .	27
4.1.6 Test 5: Sudoku generieren durch zufälliges herausnehmen von Zahlen mit Regeln . . . . .	30
4.2 Generator zum erstellen von Sudokus . . . . .	33
<b>5 Fazit</b>	<b>36</b>
<b>6 Literaturverzeichnis</b>	<b>36</b>

# Abbildungsverzeichnis

1 Notizen in Online-Sudoku . . . . .	5
2 Felder die beeinflusst werden . . . . .	7
3 Zweite Hinreichende Bedingung . . . . .	9
4 Problemsituation . . . . .	11
5 Problemsituation mit Erläuterung . . . . .	11
6 Naked-Subset . . . . .	13
7 Force-Chain . . . . .	14
8 Test 1: Wahrscheinlichkeiten Sudokus bestimmter Lösbarkeit zu generieren . . . . .	24
9 Test 4: Wahrscheinlichkeiten Sudokus bestimmter Lösbarkeit zu generieren . . . . .	29
10 Vergleich von Test 2, Test 3 und Test 4 . . . . .	29
11 Unavoidable Squares . . . . .	30

## Tabellenverzeichnis

1	Test 1: Sudoku generieren durch zufälliges herausnehmen von Zahlen . . . . .	23
2	Test 3: Sudoku generieren durch zufälliges Hinzufügen mit mäßigen Regel . . . . .	26
3	Test 4: Sudoku generieren durch zufälliges Hinzufügen mit strengen Regel . . . . .	28

# **1 Einleitung**

Sudokus sind Logikrätsel die 1979 von Howard Garns erfunden und 1984 in Japan unter dem heutigen Namen populär wurden. Im Westen begann die Verbreitung des Sudokus mit einer erstmaligen Veröffentlichung in der New York Times im November 2004. Bei dem Rätsel selbst handelt es sich um Logikrätsel, die aus den lateinischen Quadraten entstanden sind. Ziel des Rätsels ist es, in einem 9x9 Quadrat in jede Zeile, jede Spalte und jedes 3x3 Quadrat die Zahlen von 1 bis 9 einzutragen. Ein "gültiges" Sudoku hat dabei aber immer nur eine Lösung. Um eine einzigartige Lösung zu besitzen, muss ein Sudoku mindestens 17 Hinweise haben, während Zeitungen und Magazine oft Sudokus mit 25 Hinweisen abdrucken. Unsere Motivation uns in unserem Projekt mit Sudokus zu beschäftigen, war, dass Sudokus die Art von Logikrätsel sind, die ein Computer viel besser lösen kann als ein Mensch. Zudem sollen Sudokus das Gehirn anregen, also kann dass schreiben zum lösen aller Sudokus auch nicht schaden.

## **2 Vorhaben/Ziele**

### **2.1 Allgemein**

Unser Projekt beschäftigt sich mit dem Lösen und Generieren von Sudokus mit Hilfe von Computern.

### **2.2 Sudoku-Lösungsprogramm**

Das Sudoku-Lösungsprogramm soll nicht nur zufällig Zahlen einsetzen, sondern ähnlich wie ein Mensch, logisch die gestellten Probleme angehen. Sollte es jedoch nicht möglich sein, nur mit Logik fortzusetzen, soll auch auf Trail-and-Error zurückgegriffen werden. Zudem soll es dem Programm möglich sein, ein nicht lösbares, sowie ein Sudoku mit mehreren Lösungen zu identifizieren.

### **2.3 Sudoku-Generator**

Zudem untersuchen wir verschiedene Arten Sudokus zu generieren. Als da wären zum Beispiel, dass zufällige einsetzen von Hinweisen in ein Leeres Sudoku oder auch das entnehmen von Zahlen aus einem bereits gelösten Sudoku. Bei unseren Untersuchungen betrachten wir vor allem ob ein generiertes Sudoku einzigartig-lösbar sind und wie lange es dauert ein einzigartig-lösbares Sudoku zu generieren bei verschiedenen Methoden.

### 3 Sudoku-Lösungsprogramm

#### 3.1 Entstehung

##### 3.1.1 Leeres Sudoku

Zunächst betrachten wir das Sudoku-Lösungsprogramm. Im Folgenden erläutern wir unser Vorgehen beim Schreiben des Codes, wir erklären die Probleme, sowie unsere Lösungen.

Die erste Frage die sich natürlich zuerst stellte war, in welcher Form man das Sudoku darstellt. Nach kurzer Überlegung entschieden wir uns das Sudoku als numpy-array darzustellen. Dabei sollte eine Zahl im numpy-array in Position und Zahl mit einem Kästchen des Sudokus übereinstimmen soll. So entspricht also ein leeres Sudoku also zunächst einem (9,9)-numpy-array. Da ein numpy-array ja nicht so einfach an einer Stelle leer sein kann und die Null in einem Sudoku keine weitere Verwendung findet, entschieden wir uns einem Leeren Feld im Sudoku die Null zuzuordnen.

Beim Lösen eines Sudokus ist es wichtig sich merken zu können, in welches Feld welche Zahlen noch Platz finden. Viele Sudoku-Programme im Internet bieten auch die Möglichkeit sich Notizen zu machen, welche Zahlen noch in ein Feld eingesetzt werden können.

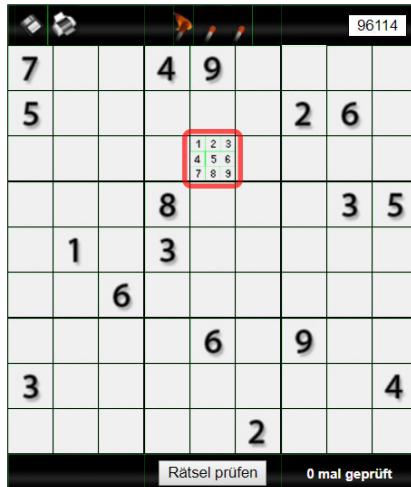


Abbildung 1: Notizen in Online-Sudoku

Natürlich kann man dem Computer nicht einfach sagen, dass er sich Notizen zu machen, sondern muss dem Computer eine Möglichkeit geben diese Informationen zu speichern. Wir haben uns entschieden dazu eine Option von numpy-arrays auszunutzen. In numpy können nämlich drei-dimensionale arrays erstellt werden. Diese haben nicht nur eine Höhe und eine Breite, sondern auch eine Tiefe. Man kann es sich vorstellen als eine Reihe von zwei-dimensionalen arrays die man hintereinander aufgestellt hat. Bei der Ausgabe sieht ein solches drei-dimensionales array etwa so aus:

```

import numpy as np

a = np.zeros((3,4,2))
a[1,:,:] = 1
a[2,:,:] = 2
print(a)
>>
[[[ 0.  0.]
  [ 0.  0.]
  [ 0.  0.]
  [ 0.  0.]]

[[ 1.  1.]
  [ 1.  1.]
  [ 1.  1.]
  [ 1.  1.]]

[[ 2.  2.]
  [ 2.  2.]
  [ 2.  2.]
  [ 2.  2.]]]
```

So nutzen wir diese Eigenschaft für unser Sudoku. Das eigentliche Sudoku bildet die 'oberste' Ebene, danach bildet die nächste Ebene, eine Ebene für die Möglichen Felder in denen eine Eins stehen kann, die darauf Folgenden eine für die in denen eine Zwei stehen kann und so weiter. Steht in einem Feld in der dritten Ebene (also die Ebene für die Zweien) eine Zwei, so bedeutet das, dass in dem eigentlichen Feld an dieser Stelle eine Zwei stehen könnte. Steht an der selben Stelle eine Null, so bedeutet das, dass in dem betreffenden Sudoku an dieser Stelle keine Zwei stehen kann. Die Funktion die so ein leeres Sudoku in dem noch alles möglich ist, realisiert, sieht wie folgt aus:

```

import numpy as np

def empty_sudoku():
    sudoku = np.zeros((10,9,9))
    for i in range(10):
        sudoku[i,:,:] = i

    return(sudoku)
```

### 3.1.2 Einträge im Sudoku verändern

Der nächste Schritt war es, eine Funktion zu schreiben mit der wir Einträge in unserem Sudoku verändern konnten. Das Problem dabei ist allerdings, dass das Setzen einer Zahl im Sudoku Einfluss darauf hat, welche Zahlen in andere Felder des Sudoku kommen können. Nach den Regeln für ein Sudoku darf in jeder Zeile, in jeder Spalte und in jedem 3x3 Kästchen jede Zahl von 1 bis 9 nur einmal auftreten. Somit beeinflusst das Setzen einer Zahl die Möglichkeiten in Folgenden Feldern:

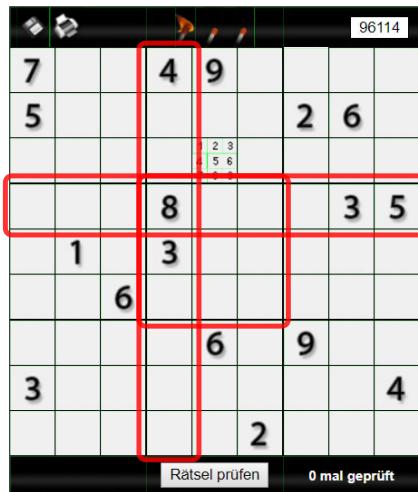


Abbildung 2: Felder die beeinflusst werden

In dem Vorliegenden Beispiel kann, aufgrund der Acht, weder in der vierten Zeile, Spalte oder dem Mittleren 3x3 Kästchen eine Acht stehen. Außerdem kann natürlich auch im Feld in dem die Acht steht, keine andere Zahl mehr stehen. Die Funktion die das Sudoku entsprechend ändert, sieht wie folgt aus:

```
def clues(sudoku, zahl, x, y):
    x = x - 1
    y = y - 1
    sudoku[0,y,x] = zahl
    sudoku[1:,y,x] = 0
    sudoku[zahl,:,:x] = 0
    sudoku[zahl,y,: ] = 0

    if x in [0,1,2]:
        if y in [0,1,2]:
            sudoku[zahl,0:3,0:3] = 0

    elif y in [3,4,5]:
        sudoku[zahl,3:6,0:3] = 0

    elif y in [6,7,8]:
        sudoku[zahl,6:9,0:3] = 0
```

```

    elif x in [3,4,5]:
        if y in [0,1,2]:
            sudoku[zahl,0:3,3:6] = 0

    elif y in [3,4,5]:
        sudoku[zahl,3:6,3:6] = 0

    elif y in [6,7,8]:
        sudoku[zahl,6:9,3:6] = 0

    elif x in [6,7,8]:
        if y in [0,1,2]:
            sudoku[zahl,0:3,6:9] = 0

        elif y in [3,4,5]:
            sudoku[zahl,3:6,6:9] = 0

        elif y in [6,7,8]:
            sudoku[zahl,6:9,6:9] = 0

    return(sudoku)

```

Durch diese Funktion wird in das Feld des Sudokus zunächst die gefragten Zahl eingesetzt. Dann wird das Feld auf allen darunterliegenden Ebenen gleich Null gesetzt, da in dieses Feld nun keine andere Zahl mehr eingesetzt werden kann. Dann werden auf der Ebene der Zahl die in das Feld eingesetzt wurde, die Zeile und Spalte in der das Feld liegt, gleich Null gesetzt. In dem Beispiel von zuvor, würde auf der Ebene in der die Möglichkeiten für die Zahl Acht gespeichert sind, also der neunten, die vierte Zeile und die vierte Spalte gleich Null gesetzt. Zuletzt wird identifiziert in welchem 3x3 Feld sich das Feld befindet, dieses wird dann ebenfalls auf der Ebene der Zahl gleich Null gesetzt.  
Mit dieser Funktion kann nun ein Sudoku wie das aus dem Beispiel erstellt werden.

### 3.1.3 Ausschluss-Prinzip für ein Feld

Wir können nun also ein Sudoku, welches wir lösen wollen, eingeben. Nun ging es daran, einen Lösungsalgorithmus zu entwickeln. Dazu mussten wir uns überlegen, welche Hinreichenden Bedingungen erfüllt sein müssen, damit eine Zahl in das Sudoku eingesetzt werden darf. Zunächst natürlich: Eine Zahl kann eingesetzt werden, wenn diese Zahl die einzige Möglichkeit ist, ein Feld zu füllen. in unserem Algorithmus haben wir diese Bedngung wie folgt umgesetzt:

```

for i in range(9):
    for j in range(9):
        reihe = sudoku[1:10,i,j]
        pruef = check_list(reihe)
        if pruef == 1:
            sudoku = clues(sudoku,int(reihe[np.argmax(reihe)]),j+1,i+1)

```

Es wird über alle Felder iteriert und bei jedem Feld alle Möglichkeiten in eine Liste geschrieben. Die Funktion `check_list` überprüft dann, wie viele Elemente in der Liste der Möglichkeiten ungleich Null sind. Sollte nur eines der Elemente ungleich Null sein, soll diese Zahl im Sudoku an der entsprechenden Stelle als neuer Hinweis mithilfe der Funktion `clues` eingefügt werden. Die dabei benutzte Funktion `check_list` ist die folgende:

```

def check_list(reihe):
    x = 0
    for i in range(9):
        if reihe[i] != 0:
            x = x + 1
    return(x)

```

### 3.1.4 Ausschluss-Prinzip für ein Kästchen

Die nächste Bedingung die wir implementierten, ist die, dass wenn in einem 3x3 Kästchen eine Zahl nur in einem Kästchen auftauchen kann, dass diese an dieser Stelle in das Sudoku eingesetzt wird. Zur Veranschaulichung:

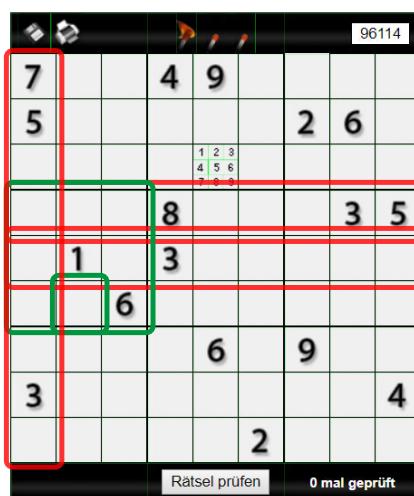


Abbildung 3: Zweite Hinreichende Bedingung

In dem großen grünen Feld kann aufgrund der anderen dreien im Sudoku nur das kleine grüne Feld im großen grünen Feld mit der drei besetzt werden. Da eine drei in das große grüne Feld rein muss, können wir die drei in das kleine grüne Feld einsetzen. Der Mechanismus der identifiziert wann so eine Zahl in das Sudoku eingesetzt werden kann, sieht so aus:

```

for n in range(1,10):
    for m in range(3):
        for o in range(3):
            square = sudoku[n,3*m:3*(m+1),3*o:3*(o+1)]
            pruef = check_square(square)
            if pruef == 1:
                if int(np.argmax(square)) in [0,1,2]:
                    y = 1

                elif int(np.argmax(square)) in [3,4,5]:
                    y = 2

                elif int(np.argmax(square)) in [6,7,8]:
                    y = 3

```

```

if int(np.argmax(square)) in [0,3,6]:
    x = 1

elif int(np.argmax(square)) in [1,4,7]:
    x = 2

elif int(np.argmax(square)) in [2,5,8]:
    x = 3

sudoku = clues(sudoku,n,(x + (3*o)),(y + (3*m)))

```

Es wird zunächst über die einzelnen Zahlen-Ebenen iteriert. Innerhalb dieser Zahlen-Ebenen wird dann über die einzelnen 3x3 Kästchen iteriert. Die Möglichkeiten in einem 3x3 Kästchen werden dann in ein array geschrieben, bei welchem mit der check\_square Funktion überprüft wird, wie viele Elemente in dem array ungleich Null sind. Sollte nur ein Element ungleich Null sein, wird dieses an der entsprechenden Stelle im Sudoku mit der clues-Funktion eingesetzt.

### 3.1.5 Ausschluss-Prinzip für Zeilen und Spalten

Einen ähnlichen Mechanismus haben wir dann auch für die Zeilen und Spalten geschrieben. Wie auch zuvor, sollte in einer Zeile oder Spalte eine Zahl nur in einem Feld möglich sein, wird diese dort eingesetzt.

```

for n in range(1,10):
    for m in range(9):
        zeile = sudoku[n,m,:]
        pruef_zeile = check_list(zeile)
        if pruef_zeile == 1:
            sudoku = clues(sudoku,n,int(np.argmax(zeile)) + 1,m + 1)

        spalte = sudoku[:,m]
        pruef_spalte = check_list(spalte)
        if pruef_spalte == 1:
            sudoku = clues(sudoku,n,m + 1,int(np.argmax(spalte)) + 1)

```

Ähnlich wie zuvor wird zunächst über die Zahlen-Ebenen und dann über die Zeilen beziehungsweise Spalten iteriert und jedes mal überprüft, wie viele Elemente in einer Zeile/Spalte ungleich Null sind, sollte es nur eine sein, wird das eine Element an der entsprechenden Stelle eingesetzt.

### 3.1.6 Double-Ausschluss

Wir versuchten unser Programm in diesem Zustand an dem oben schon mehrmals benutzten Beispiel. Wir mussten jedoch feststellen, dass unser Programm an einem bestimmten Punkt nicht weiter kam. Wir mussten noch weitere Mechanismen hinzufügen um Sudoku wie das folgende lösen zu können:

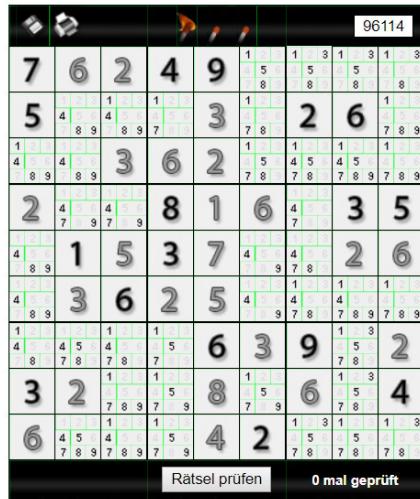


Abbildung 4: Problemsituation

Um diese Situation zu lösen, müssen wir einer besonderen Konstellation, ganz besondere Beachtung schenken. Diese sieht wie folgt aus:

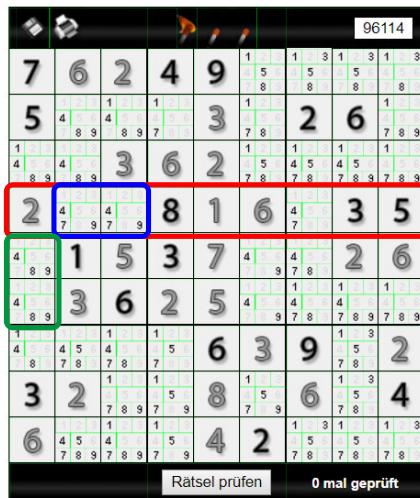


Abbildung 5: Problemsituation mit Erläuterung

Betrachten wir die rot umrandete Zeile, fällt auf das noch die Zahl neun fehlt. Des Weiteren fällt uns auf, dass die Neun nur in den Blau umrandeten Feldern sein könnte. Da in der roten Zeile eine Neun sein muss, bedeutet dies, dass in den grünen Feldern die Neun nicht sein kann. Und diese Korrelation ist es, die es ermöglicht das Sudoku zu lösen. Es lässt sich allgemein sagen, dass wenn alle Möglichkeiten eine Zahl zu setzen in einer Zeile oder Spalte nur in einem Kästchen liegen, dann können alle anderen Möglichkeiten diese Zahl in diesem Kästchen zu setzen ignoriert und somit gleich Null gesetzt werden. Die Programmierung gestaltete sich nun ein wenig anspruchsvoller. Unsere Lösung sieht wie folgt aus:

```
if run >= 3:  
    for n in range(1,10):  
        for m in range(9):  
            zeile = sudoku[n,m,:]  
            pruef_zeile = check_list(zeile)  
            if pruef_zeile == 2 or pruef_zeile == 3:
```

```

doubles = check_double(zeile,pruef_zeile)
if doubles[0] == True:
    if m in [0,3,6]:
        sudoku[n,m+1:m+3,doubles[1]*3:(doubles[1]+1)*3] = 0

    elif m in [1,4,7]:
        sudoku[n,m-1,doubles[1]*3:(doubles[1]+1)*3] = 0
        sudoku[n,m+1,doubles[1]*3:(doubles[1]+1)*3] = 0

    else:
        sudoku[n,m-2:m,doubles[1]*3:(doubles[1]+1)*3] = 0

spalte = sudoku[:,m]
pruef_spalte = check_list(spalte)
if pruef_spalte == 2 or pruef_spalte == 3:
    doubles = check_double(spalte,pruef_spalte)
    if doubles[0] == True:
        if m in [0,3,6]:
            sudoku[n,doubles[1]*3:(doubles[1]+1)*3,m+1:m+3] = 0

        elif m in [1,4,7]:
            sudoku[n,doubles[1]*3:(doubles[1]+1)*3,m-1] = 0
            sudoku[n,doubles[1]*3:(doubles[1]+1)*3,m+1] = 0

        else:
            sudoku[n,doubles[1]*3:(doubles[1]+1)*3,m-2:m] = 0

```

### 3.1.7 Naked-Subset

Zunächst dachten wir, dass wir damit ein Programm geschrieben hätten, welches alle Sudokus mit einer einzigen Lösung lösen könne. Doch stellte sich nach einigen Versuchen heraus, dass es doch noch einige Sudokus gibt, welche eine einzige Lösung haben, aber unser Programm nicht lösen kann. Durch weitere Recherche fanden wir nun eine weitere Methode, welche zum lösen einiger schwerer Sudokus benötigt wird. Im folgenden erstmal das problematische Sudoku.

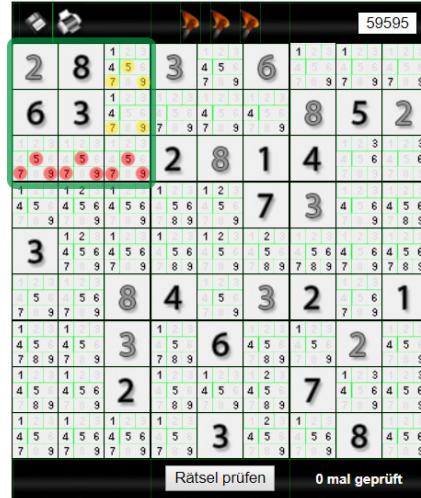


Abbildung 6: Naked-Subset

Betrachtet man das grüne 3x3 Kästchen, fällt auf, dass in den unteren drei Feldern nur die rot markierten Möglichkeiten untergebracht werden können. Daraus folgt, dass diese Rot markierten Möglichkeiten auch nur in diesen drei Feldern untergebracht werden können. Somit können die gelb markierten Möglichkeiten eliminiert werden. Der Code der solche Situationen identifiziert sieht wie folgt aus:

```

for a in range(3):
    for b in range(3):
        square = sudoku[:,a*3:(a+1)*3,b*3:(b+1)*3]
        for c in range(3):
            for d in range(3):
                if square[0,c,d] == 0:
                    compare = square[:,c,d]
                    num = check_list(compare[1:10])
                    sims = 0
                    coor = []
                    elem = []
                    for i in range(3):
                        for j in range(3):
                            if square[0,i,j] == 0:
                                test = square[:,i,j]
                                if np.array_equal(compare, test) == True:
                                    sims = sims + 1
                                    coor.append((i,j))

                    if num == sims:
                        for k in range(10):
                            if compare[k] != 0:
                                elem.append(compare[k])

                    for i in range(3):
                        for j in range(3):
                            if (i,j) not in coor and square[0,i,j] == 0:
                                for n in elem:
                                    sudoku[int(n),int((a*3)+i),int((b*3)+j)] = 0

```

Er funktioniert nach dem Prinzip, dass zunächst alle 3x3 Kästchen durchgegangen werden, in diesen Kästchen werden alle Felder und ihre Möglichkeiten durchgegangen. und mit allen anderen Feldern im 3x3 Kästchen verglichen. Sollte es Felder mit den gleichen Möglichkeiten gibt, wird gezählt ob die Anzahl der Felder mit der Anzahl der Möglichkeiten übereinstimmen, dann werden diese Möglichkeiten aus den anderen Feldern entfernt.

### 3.1.8 Force-Chain

Aber nur mit dieser Funktion ließ sich das problematische Sudoku nicht lösen. Allerdings fanden wir auch keine Methoden um die Möglichkeiten im Sudoku einzuschränken. Wir fanden durch weitere Recherche heraus, dass es Sudokus gibt, bei denen man nur mit konventionellen Methoden nicht weiter kommt, sodass man sogenannte Force-Chains nutzen muss. Dabei sucht man sich ein leeres Feld, welches mit möglichst vielen anderen Feldern in Verbindung steht und welches nur noch zwei bis drei Möglichkeiten besitzt. Dann setzt man in dieses Feld eine der Möglichkeiten ein und versucht das Sudoku zu lösen. Kommt man zu einem Ergebnis, war die Auswahl richtig, kommt man aber zu einem Widerspruch, war die Auswahl wohl falsch und man kann eine der anderen Möglichkeiten probieren. Bei unserem problematischen Sudoku boten sich beispielsweise die folgenden markierten Felder an:



Abbildung 7: Force-Chain

Auch hier war die Implementierung etwas problematisch. Die entgültige Lösung nun hier:

```
if run >= 2:
    for y in range(9):
        for x in range(9):
            if sudoku[0,y,x] == 0:
                reihe = sudoku[1:,y,x]
                trys = check_list(reihe)
                if trys == 2:
                    elem = []
                    for t in range(9):
                        if reihe[t] != 0:
                            elem.append(reihe[t])

    first_try = int(elem[0])
```

```

second_try = int(elem[1])
TE_sudoku = np.copy(sudoku)
TE_sudoku = clues(TE_sudoku,first_try,x+1,y+1)
result_one = SudokuSolve(TE_sudoku)
if result_one[0] == 'Zwei Lösungen':
    return(result_one)

TE_sudoku = np.copy(sudoku)
TE_sudoku = clues(TE_sudoku,second_try,x+1,y+1)
result_two = SudokuSolve(TE_sudoku)
if result_one[0] == 'Zwei Lösungen' or result_two[0] == 'Zwei Lösungen':
    return(['Zwei Lösungen',result_two[1]])
elif result_one[0] == result_two[0]:
    if result_one[0] == 'Eine Lösung':
        if np.array_equal(result_one[1], result_two[1]):
            return(result_one)
        else:
            return('Zwei Lösungen',result_one[1])
    elif result_one[0] == 'Keine Lösung':
        return(result_one)

elif result_one[0] == 'Eine Lösung' and result_two[0] == 'Keine Lösung':
    return(result_one)

else:
    return(result_two)

for y in range(9):
    for x in range(9):
        if sudoku[0,y,x] == 0:
            reihe = sudoku[1:,y,x]
            trys = check_list(reihe)
            if trys == 3:
                elem = []
                for t in range(9):
                    if reihe[t] != 0:
                        elem.append(reihe[t])

                first_try = int(elem[0])
                second_try = int(elem[1])
                third_try = int(elem[2])
                TE_sudoku = np.copy(sudoku)
                TE_sudoku = clues(TE_sudoku,first_try,x+1,y+1)
                result_one = SudokuSolve(TE_sudoku)
                if result_one[0] == 'Zwei Lösungen':
                    return(result_one)

                TE_sudoku = np.copy(sudoku)
                TE_sudoku = clues(TE_sudoku,second_try,x+1,y+1)
                result_two = SudokuSolve(TE_sudoku)
                if result_two[0] == 'Zwei Lösungen':
                    return(result_two)

```

```

TE_sudoku = np.copy(sudoku)
TE_sudoku = clues(TE_sudoku,third_try,x+1,y+1)
result_three = SudokuSolve(TE_sudoku)
results = [result_one[0], result_two[0], result_three[0]]
ones = []
if 'Zwei Lösungen' in results:
    if result_one[0] == 'Zwei Lösungen':
        return(['Zwei Lösungen',result_one[1]])

    elif result_two[0] == 'Zwei Lösungen':
        return(['Zwei Lösungen',result_two[1]])

    else:
        return(['Zwei Lösungen',result_three[1]])

elif 'Eine Lösung' in results:
    for n in range(2):
        if results[n] == 'Eine Lösung':
            ones.append(n)

    if len(ones) == 1:
        if ones[0] == 0:
            return(result_one)

        elif ones[0] == 1:
            return(result_two)
        else:
            return(result_three)

    elif len(ones) == 2:
        if 0 in ones:
            if 1 in ones:
                if np.array_equal(result_one[1], result_two[1]) == True:
                    return(result_one)

                else:
                    return(['Zwei Lösungen',result_one[1]])
            else:
                if np.array_equal(result_one[1], result_three[1]) == True:
                    return(result_one)

                else:
                    return(['Zwei Lösungen',result_one[1]])
        else:
            if np.array_equal(result_two[1], result_three[1]) == True:
                return(result_two)
            else:
                return(['Zwei Lösungen',result_two[1]])

    else:
        if np.array_equal(result_one[1], result_two[1]) == True:
            if np.array_equal(result_one[1], result_three[1]) == True:
                return(result_one)
            else:

```

```

        return(['Zwei Lösungen',result_one[1]])

    else:
        return(['Zwei Lösungen',result_one[1]])

else:
    return(result_one)

return(['Zwei Lösungen',sudoku])

```

Wie dieser Abschnitt funktioniert ist etwas komplizierter und wird auch nochmal im Programmcode mit Kommentaren erklärt. Hier nun ein kurzer Überblick über seine Funktion. Die Methodik der Force-Chain soll erst eingesetzt werden, wenn alle anderen Möglichkeiten ausgeschöpft sind, deshalb beginnt dieser Abschnitt, mit der Bedingung, dass run größer oder gleich zwei ist, das Programm also im zweiten Durchlauf ohne Veränderung des Sudokus ist. Falls diese Bedingung erfüllt ist, sucht das Programm zunächst nach einem leeren Feld welches nur noch zwei offene Möglichkeiten besitzt. Ist ein solches Feld gefunden, wird in eine Kopie des Orginal-Sudokus die erste der beiden Möglichkeiten eingesetzt und auf die Lösungsfunktion rekursiv zugegriffen, um das Sudoku mit diesen zusätzlichen Informationen zu lösen. Sollte nun das Ergebnis sein, dass das Sudoku zwei oder mehr Lösungen hat, wird dieses auch zurückgegeben. Ist das Ergebnis jedoch, dass das Sudoku eine oder keine Lösung hat, kann das Einsetzen der anderen Möglichkeit noch zu einem anderen Ergebnis führen. So wird auch die andere Möglichkeit eingesetzt und gelöst. Zuletzt werden die Ergebnisse verglichen. Sollte eines der Ergebnisse sein, dass es zwei oder mehr mögliche Lösungen gibt, wird dies, sowie eine mögliche Lösung zurückgegeben. Sind beide Ergebnisse, dass es eine einzige Lösung gibt, so werden die Ergebnisse verglichen, sollten sie gleich sein, wird das Ergebnis, sowie die Information, dass die Lösung einzigartig ist, zurückgegeben. Wenn sie nicht gleich sind, wird zurückgegeben, dass es zwei Lösungen gibt, sowie eine der Lösungen. Sollte nur eine der beiden Ergebnisse eine Lösung sein, so wird diese zurückgegeben, sollte keines der beiden Ergebnisse eine Lösung sein, wird zurückgegeben, dass es keine Lösung gibt. Nun ist dies aber nicht der einzige Teil der Force-Chain. Sollte der unwahrscheinliche Fall eintreten, dass es kein leeres Feld mit zwei oder weniger Möglichkeiten gibt, wird ein Feld mit drei Möglichkeiten gesucht und das gesamte Prozedere ähnlich zu dem Fall mit zwei Möglichkeiten durchgeführt. Wenn es kein Feld mit drei oder weniger Möglichkeiten gibt, bedeutet das, dass das Sudoku mit hoher Wahrscheinlichkeit weniger als siebzehn besetzte Felder hat und somit zwei oder mehr Lösungen hat. Aus diesem Grund haben wir uns entschieden, in so einem Fall einfach ausgeben zu lassen, dass das Sudoku zwei oder mehr Lösungen hat. Somit haben wir ein Programm, welches in jedem Fall ein Ergebnis liefert und damit unser erstes Ziel erfüllt.

### 3.2 Gelöste Rätsel

Hier nun eine Liste von Sudokus mit denen wir unser Programm überprüft haben. Zum Zwecke der Wiederholbarkeit, haben wir Sudokus von der Internet-Seite <http://www.sudoku17.de>. Dort hat jedes Sudoku eine eigene Nummer, mit der es immer wieder aufgerufen werden kann. So stellen wir hier eine Liste der Nummern der Sudokus auf, an denen wir unser Programm getestet haben, alle davon rangieren im höchsten Schwierigkeitsgrad.

- 71230
- 7315
- 59595
- 67806
- 60216
- 59415
- 79393
- 59818

## 4 Sudoku-Generator

Nachdem wir uns zuvor mit dem Lösen von Sudokus beschäftigt haben, untersuchen wir nun verschiedene Arten Sudokus zu generieren. Zum einen die Variante, in ein leeres Sudoku zufällig Hinweise einzusetzen und zum anderen, aus einem bereits gelösten Sudoku zufällig Zahlen wegzunehmen bis die gewünschte Zahl an Hinweisen erreicht ist. Besonderes Augenmerk richten wir bei unserer Untersuchung auf die Lösbarkeit der generierten Sudokus. Vorarbeit wurde in dieser Hinsicht schon geleistet, da Gary McGuire, Bastian Tugemann und Gilles Civario 2013 in ihrer Arbeit 'There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration'" bewiesen haben, dass alle lösbarren Sudokus mit 16 oder weniger Hinweisen mindestens zwei Lösungen haben. Das beschränkt die Hinweiszahlen die wir untersuchen müssen auf 17 bis 81 Hinweise.

### 4.1 Generierungsprozesse und Experimente

#### 4.1.1 Generierung vollständiger Sudokus

Zunächst befassten wir uns mit der Generierung von vollständigen Sudokus. Hier war unsere erste Idee, der Reihe nach zufällig Zahlen in das Sudoku Feld einzusetzen bis das Einsetzen weiterer Zahlen zu Widersprüchen führen würde, oder das Sudoku vollständig ist. Sollte es zu Widersprüchen kommen, sollte die Funktion zurückgehen und bereits gesetzte Zahlen neu wählen, bis alle Felder gefüllt sind. Dies erwies sich jedoch in der Umsetzung schwierig. Als zweckdienlicher stellte sich das Verfahren heraus, Zahlen nach gegebenen Möglichkeiten einzusetzen und Widersprüche zu überspringen. Erzeugt man so nur genug Sudokus, werden auch schnell einige Auftreten, die keine Widersprüche aufweisen und somit als vollständige Sudokus verwendet werden können. Der Code der dies bewerkstelligt sieht so aus:

```
def mkAnswer(sudoku):
    for Y in range(9):
        for X in range(9):
            if sudoku[Y,X] == 0:
                sudoku = setNumber(X,Y,sudoku)

    return(sudoku)

def setNumber(X,Y,sudoku):
    possibilitys = mklst(X,Y,sudoku)
    P = len(possibilitys)
    if P > 0:
        sudoku[Y,X] = np.random.choice(possibilitys,1)

    return(sudoku)
```

```

def mklst(X,Y,sudoku):
    lst = [1,2,3,4,5,6,7,8,9]
    zeile = sudoku[Y,0:9]
    spalte = sudoku[0:9,X]

    if Y in [0,1,2]:
        if X in [0,1,2]:
            square = sudoku[0:3,0:3]

        elif X in [3,4,5]:
            square = sudoku[0:3,3:6]

        elif X in [6,7,8]:
            square = sudoku[0:3,6:9]

    elif Y in [3,4,5]:
        if X in [0,1,2]:
            square = sudoku[3:6,0:3]

        elif X in [3,4,5]:
            square = sudoku[3:6,3:6]

        elif X in [6,7,8]:
            square = sudoku[3:6,6:9]

    elif Y in [6,7,8]:
        if X in [0,1,2]:
            square = sudoku[6:9,0:3]

        elif X in [3,4,5]:
            square = sudoku[6:9,3:6]

        elif X in [6,7,8]:
            square = sudoku[6:9,6:9]

    rechteck = np.reshape(square,9)

    for s in spalte:
        if s in lst:
            lst.remove(s)

    for z in zeile:
        if z in lst:
            lst.remove(z)

    for r in rechteck:
        if r in lst:
            lst.remove(r)

    return(lst)

```

```

solved = False
while solved != True:
    sudoku = np.zeros((9,9))
    sudoku = mkAnswer(sudoku)
    solved = check_sudoku(sudoku)

```

Zunächst die mkAnswer-Funktion: Diese gibt das Sudoku aus. Es wird über alle Felder eines leeren Sudokus iteriert und mit der Funktion setNumber in die leeren Felder Zahlen eingesetzt. Bei der setNumbers Funktion wird zunächst eine Liste der Möglichen Zahlen, die in ein Feld kommen kann, mit Hilfe der mklst Funktion erstellt. Sollte die Liste nicht leer sein, wird zufällig eine der Möglichkeiten eingesetzt. Wenn die Liste Leer ist, bleibt in dem Feld eine Null. Bei der mklst Funktion wird zunächst eine vollständige Liste der möglichen Zahlen erstellt. Aus dieser werden dann die Zahlen entfernt, wenn sie in der Zeile, Spalte oder dem 3x3 Kästchen schon einmal vorkommen. Zuletzt werden auf diese Variante Sudokus erstellt, bis das Programm ein Sudoku erstellt hat, in dem keine Nullen mehr sind. Überprüft wird dies mit der check\_sudoku Funktion die schon bei unserem Sudoku-Lösungsprogramm zum Einsatz kam. Am Ende hat man eine vollständig ausgefülltes Sudoku.

#### 4.1.2 Test 1: Sudoku generieren durch zufälliges herausnehmen von Zahlen

Nachdem wir nun ein vollständiges Sudoku haben, versuchen wir nun durch zufälliges herausnehmen von Zahlen aus ebendiesem ein Sudoku-Puzzle zu erstellen. Für unseren Test haben wir unser Programm für alle möglichen Hinweiszahlen von 17 bis 81 je 500 Sudoku-Puzzle (also Sudokus in denen nur die Hinweise stehen) generiert und auf ihre Lösbarkeit getestet. Mögliche Ergebnisse waren dabei, "keine Lösung", "eine einzige Lösung" oder "zwei oder mehr Lösungen". Der Code für diesen Test ist folgender:

```

def make_sudoku(N):
    pos = [0,1,2,3,4,5,6,7,8]
    z = 0
    solved = False
    while solved != True:
        sudoku = np.zeros((9,9))
        sudoku = mkAnswer(sudoku)
        solved = check_sudokuG(sudoku)

    x = check_list(sudoku.reshape(81))
    while x != N:
        i = np.random.choice(pos)
        j = np.random.choice(pos)
        sudoku[i,j] = 0
        x = check_list(sudoku.reshape(81))

    return(sudoku)

```

```

ergebnisse = []
for N in range(17,82):

    null = 0
    eins = 0
    zwei = 0

    for n in range(500):
        quest = make_sudoku(N)
        sudoku = empty_sudoku()
        for y in range(1,10):
            for x in range(1,10):
                if quest[y-1,x-1] != 0:
                    sudoku = clues(sudoku,int(quest[y-1,x-1]),x,y)

SUSO = SudokuSolve(sudoku)

if SUSO[0] == 'Keine Lösung':
    null = null + 1

elif SUSO[0] == 'Zwei Lösungen':
    zwei = zwei + 1

elif SUSO[0] == 'Eine Lösung':
    eins = eins + 1

ergebnisse.append([null,eins,zwei])

```

Die Funktion `make_sudoku` erstellt wie zuvor beschrieben ein Sudoku mit  $N$  Hinweisen und der dar-auffolgende Code testet die einzelnen Sudokus auf ihre Lösbarkeit. Das Ergebnis ist in der folgenden Tabelle festgehalten.

Hinweise	Eine Loesung	Zwei oder mehr Loesungen
17	0	500
18	0	500
19	0	500
20	0	500
21	0	500
22	0	500
23	0	500
24	0	500
25	0	500
26	0	500
27	1	499
28	2	498
29	1	499
30	6	494
31	12	488
32	19	481
33	32	468
34	42	458
35	56	444
36	72	428
37	110	390
38	133	367
39	139	361
40	155	345
41	192	308
42	194	306
43	261	239
44	258	242
45	281	219
46	304	196
47	337	163
48	350	150
49	390	110
50	383	117
51	409	91
52	415	85
53	430	70
54	432	68
55	437	63
56	455	45
57	455	45
58	476	24
59	472	28
60	481	19
61	484	16
62	487	13
63	485	15
64	493	7
65	494	6
66	496	4
67	497	3
68	498	2
69	498	2
70	497	3
71	499	1
72	499	1
73	500	0
74	500	0
75	500	0
76	500	0
77	500	0
78	500	0
79	500	0
80	500	0
81	500	0

Tabelle 1: Test 1: Sudoku generieren durch zufälliges herausnehmen von Zahlen

Die folgende Graphische Darstellung zeigt, wie sich die Wahrscheinlichkeiten ein Sudoku mit einer einzigen Lösung und ein Sudoku mit zwei oder mehr Lösungen zu generieren mit zunehmender Anzahl verhalten.

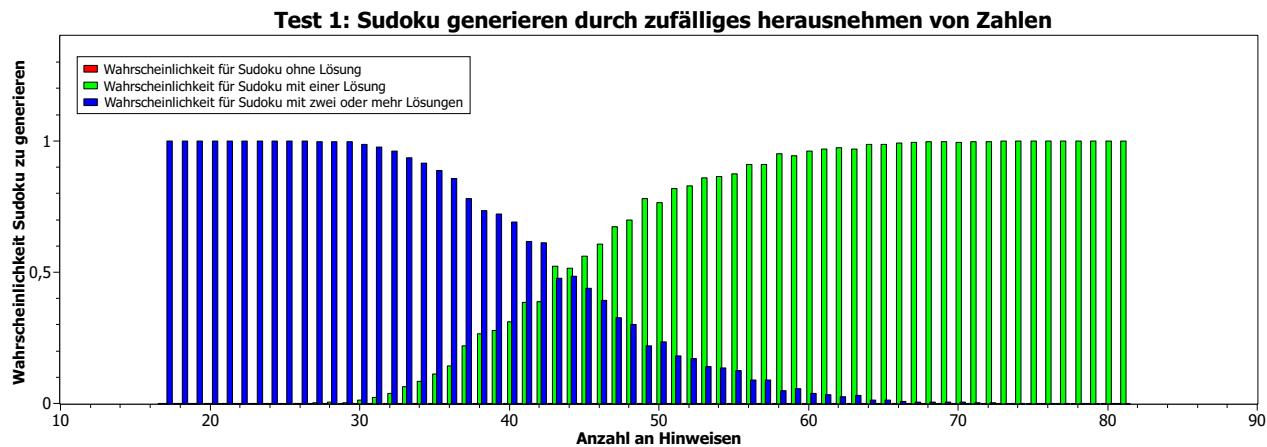


Abbildung 8: Test 1: Wahrscheinlichkeiten Sudokus bestimmter Lösbarkeit zu generieren

Die Auswertung zeigt ganz klar die Stärken dieser Methode auf. Zum einen sind alle Sudokus die wir generieren entweder einfach oder mehrfach lösbar, es kann aber nie passieren, dass eines unserer Sudokus keine Lösung aufweist. Des Weiteren fällt auf, dass die Anzahl von Sudokus mit einer einzigen Lösung nahezu logistisch zunimmt. So, müssen wir nicht allzu lange suchen, wenn wir ein einzigartig lösbares Sudoku mit 30 oder mehr Hinweisen generieren wollen. Allerdings bedeutet das auch, dass wir signifikant länger suchen müssen, wenn es darum geht ein Sudoku mit 20 oder weniger Hinweisen zu finden.

#### 4.1.3 Test 2: Sudoku generieren durch zufälliges Hinzufügen ohne Regel

Das nächste Verfahren, welches wir untersuchen wollten, war: Das Generieren von Sudokus durch zufälliges Einsetzen von Zahlen in ein leeres Sudoku. Das allgemeine Verfahren blieb dabei im Vergleich zum vorigen Test das Gleiche. Deswegen hier nur das Stück Code, welches das Sudoku erstellt:

```
pos = [1,2,3,4,5,6,7,8,9]
sudoku = empty_sudoku()
c = 0
while c < N:
    x = np.random.choice(pos) - 1
    y = np.random.choice(pos) - 1
    z = np.random.choice(pos)
    if sudoku[0,y,x] == 0:
        sudoku = clues(sudoku,z,x + 1,y + 1)
        c = c + 1
```

Wie wir erwartet haben, führt unregulierte einsetzen von Zufallszahlen nur dazu, dass wir widersprüchliche Sudoku-Puzzle bekommen. Wie eindeutig dies aber schon bei geringen Anzahlen von Hinweisen ist, hat uns doch überrascht.

#### 4.1.4 Test 3: Sudoku generieren durch zufälliges Hinzufügen mit mäßigen Regel

Als nächstes haben wir uns damit beschäftigt, dass vorangegangene Verfahren zu verbessern. Eine Bedingung, damit ein Sudoku eindeutig lösbar ist, die wir durch Recherche herausgefunden haben, war, dass mindesten 8 von 9 Zahlen im Sudoku verwendet worden sein müssen. Aus diesem Grund haben wir den nächsten Generator so programmiert, dass die ersten acht Zahlen die er einsetzt alle verschieden sind:

```
korrigiert = [0,1,2,3,4,5,6,7,8]
pos = [1,2,3,4,5,6,7,8,9]
sudoku = empty_sudoku()
c = 0
while c < N:
    x = np.random.choice(korrigiert)
    y = np.random.choice(korrigiert)
    z = np.random.choice(pos)
    if sudoku[0,y,x] == 0:
        sudoku = clues(sudoku,z,x + 1,y + 1)
        c = c + 1
        if c < 8:
            pos.remove(z)

    else:
        pos = [1,2,3,4,5,6,7,8,9]
```

Wir haben nicht viel erwartet, aber das Ergebnis hat uns überrascht. Anstatt 16 Sudokus mit zwei oder mehr Lösungen, gab uns dieser Generator vierzig. Die Verteilung dieser sieht man in der folgenden Tabelle:

Hinweise	Keine Lösung	Zwei oder mehr Lösungen
17	482	18
18	490	10
19	494	6
20	497	3
21	498	2
22	500	0
23	500	0
24	499	1
25	500	0
26	500	0
27	500	0
28	500	0
29	500	0
30	500	0
31	500	0
32	500	0
33	500	0
34	500	0
35	500	0
36	500	0
37	500	0
38	500	0
39	500	0
40	500	0
41	500	0
42	500	0
43	500	0
44	500	0
45	500	0
46	500	0
47	500	0
48	500	0
49	500	0
50	500	0
51	500	0
52	500	0
53	500	0
54	500	0
55	500	0
56	500	0
57	500	0
58	500	0
59	500	0
60	500	0
61	500	0
62	500	0
63	500	0
64	500	0
65	500	0
66	500	0
67	500	0
68	500	0
69	500	0
70	500	0
71	500	0
72	500	0
73	500	0
74	500	0
75	500	0
76	500	0
77	500	0
78	500	0
79	500	0
80	500	0
81	500	0

Tabelle 2: Test 3: Sudoku generieren durch zufälliges Hinzufügen mit mäßigen Regel

#### 4.1.5 Test 4: Sudoku generieren durch zufälliges Hinzufügen mit strengen Regel

Zuletzt haben wir die Bedingung hinzugefügt, dass beim setzen einer Zahl darauf geachtet wird, dass die Zahl in der Spalte, Zeile und dem 3x3 Kästchen noch nicht vorkommt. Sollte auf diese Weise keine Zahl gesetzt werden können, so soll das Programm von vorne Beginnen. Auf diese Weise haben wir uns erhofft, viel weniger Sudokus ohne Lösung und unter Umständen vielleicht auch welche mit die eindeutig Lösbar sind

```
pos = [1,2,3,4,5,6,7,8,9]
sudoku = empty_sudoku()
c = 0
pairs = [[x,y] for x in range(9) for y in range(9)]
shuffle(pairs)
while c < N:
    koor = pairs[0]
    x = koor[0]
    y = koor[1]
    z = np.random.choice(pos)

    pruef = mklstP(x,y,sudoku)
    if sudoku[0,y,x] == 0 and z in pruef:
        sudoku = clues(sudoku,z,x + 1,y + 1)
        c = c + 1
        pairs.remove(koor)
        if c < 8:
            pos.remove(z)

    else:
        pos = [1,2,3,4,5,6,7,8,9]

elif sudoku[0,y,x] == 0 and pruef == []:
    sudoku = empty_sudoku()
    c = 0
    pairs = [[x,y] for x in range(9) for y in range(9)]
    shuffle(pairs)
```

Das Ergebnis hat unsere Erwartungen erfüllt in der Hinsicht, dass viel mehr Sudokus mit zwei oder mehr Lösungen generiert wurden, sowie einige mit einer eindeutigen Lösung. Allerdings haben sich auch einige andere Probleme offenbart. So braucht dieser Generator für zunehmende Anzahl von Hinweisen signifikant länger, nur um ein zu überprüfendes Sudoku zu generieren, weshalb wir den Test bei 64 Hinweisen abgebrochen haben. Nun mag es eigenartig erscheinen, dass der Computer zum generieren von vollständigen Sudokus viel weniger Zeit braucht, ob wohl er bei beiden eine ähnliche Technik verwendet. Der Grund dafür liegt tatsächlich in diesem kleinen Unterschied der Vorgehensweise. Bei der Generierung der vollständigen Sudokus geht der Computer Zeile für Zeile, Feld für Feld durch, dabei beeinflussen sich die gesetzt Zahlen direkt, da sie oft im selben 3x3 Kästchen und noch häufiger in der selben Zeile sind. So ist die Auswahl beim setzen einer neuen Zahl viel geringer. Anders beeinflussen sich hintereinander gesetzte Zahlen beim zufälligen einfügen fast nie, die Zahlen werden gesetzt, obwohl dies bedeutet, dass in ein anderes Feld vielleicht keine Zahl mehr gesetzt werden kann, so kommt es meistens nach 40-50 gesetzten Zahlen vor, dass der Computer ein leeres Feld findet, in welches keine Zahlen mehr kommen können.

So machte die Generierung von Sudoku-Puzzles Probleme. Aber auch das Lösen der generierten Sudokus konnte unter Umständen sehr lange dauern. Dies hing damit zusammen, dass wenn es darum geht Sudokus zu überprüfen, die keine Lösung haben, der Computer alle möglichen Kombinationen

probieren muss, bis er tatsächlich sagen kann, dass ein Sudoku keine Lösung hat. Hat ein Sudoku mehrere Lösungen oder auch nur eine, dann können oft viele Schritte der Force-Chain übersprungen werden. Hat das Sudoku allerdings keine Lösung und widerspricht sich nicht auf den ersten Blick (also, dass Beispielsweise die gleiche Zahl zweimal in einer Zeile vorkommt), dann kann es durchaus länger dauern, die Lösbarkeit eines Sudokus zu überprüfen. In der folgenden Tabelle dann nun die Ergebnisse dieses Tests.

Hinweise	Keine Loesung	Eine Loesung	Zwei oder mehr Loesungen
17	43	0	457
18	49	0	451
19	66	0	434
20	118	0	328
21	177	0	323
22	222	0	278
23	276	0	224
24	309	1	190
25	349	1	150
26	402	3	95
27	444	2	54
28	476	1	32
29	484	1	15
30	486	0	14
31	495	1	4
32	498	0	2
33	498	2	0
34	499	0	1
35	500	0	0
36	500	0	0
37	500	0	0
38	500	0	0
39	500	0	0
40	499	0	1
41	500	0	0
42	500	0	0
43	500	0	0
44	500	0	0
45	500	0	0
46	500	0	0
47	500	0	0
48	500	0	0
49	500	0	0
50	500	0	0
51	500	0	0
52	500	0	0
53	500	0	0
54	500	0	0
55	500	0	0
56	500	0	0
57	500	0	0
58	500	0	0
59	500	0	0
60	500	0	0
61	500	0	0
62	500	0	0
63	500	0	0
64	500	0	0

Tabelle 3: Test 4: Sudoku generieren durch zufälliges Hinzufügen mit strengen Regel

Zunächst eine Darstellung der Wahrscheinlichkeiten bei diesem Verfahren

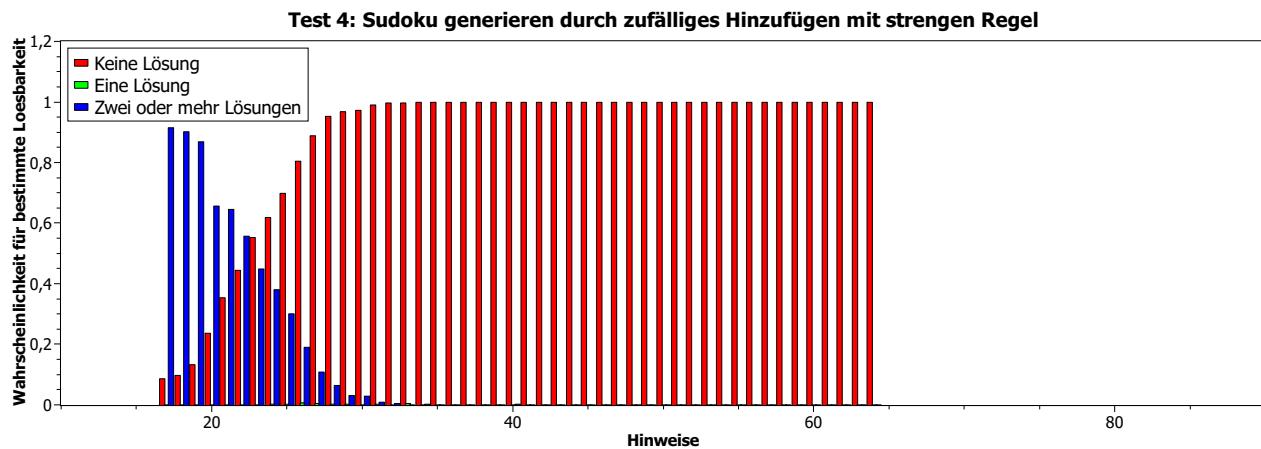


Abbildung 9: Test 4: Wahrscheinlichkeiten Sudokus bestimmter Lösbarkeit zu generieren

Und nun ein Vergleich dieser Methode mit den beiden vorigen Methoden zum zufälligen einsetzen.

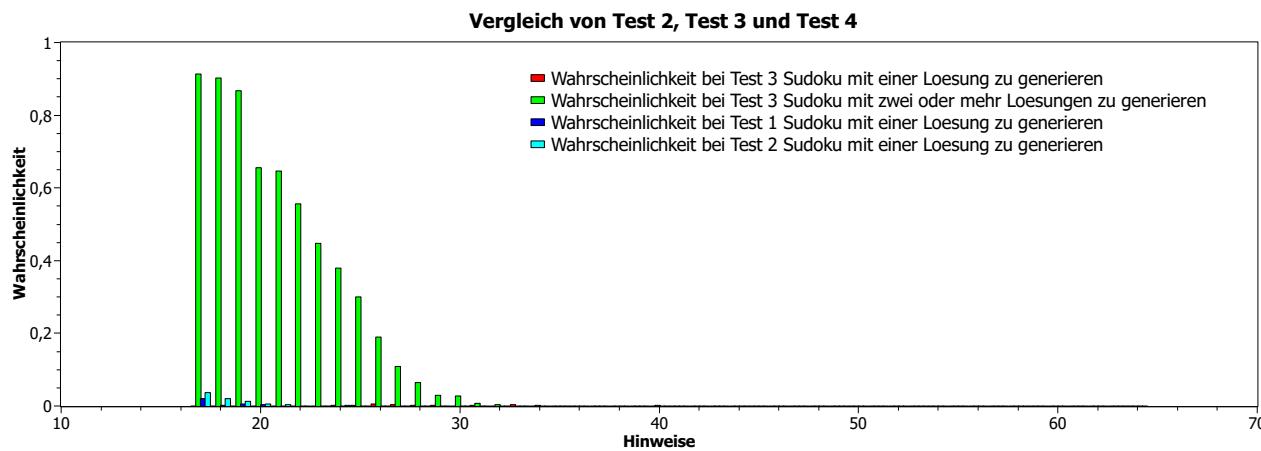


Abbildung 10: Vergleich von Test 2, Test 3 und Test 4

Es ist sehr deutlich, wie die letzte Variante mehr lösbare Sudokus hervorbringt, als die beiden zuvor. Allerdings wird genauso deutlich, dass Sudokus mit mehr als 30 Hinweisen, nur sehr selten lösbar sind. Deshalb haben wir beschlossen nun die erste Variante weiter zu verbessern.

#### 4.1.6 Test 5: Sudoku generieren durch zufälliges herausnehmen von Zahlen mit Regeln

Um die erste Version des Generators zu verbessern, mussten wir herausfinden, wann ein Sudoku zwei oder mehr Lösungen haben kann. Durch ausprobieren und selbst Sudokus lösen, fanden wir, was man "unavoidable squares" nennt. Es sind gewisse Konstellationen die man in fertigen Sudokus finden kann. Was diese Konstellationen auszeichnet, ist, dass immer mindestens eines der Felder von diesen Konstellationen gegeben sein muss, um ein eindeutig lösbares Sudoku zu erhalten. Ein Beispiel verdeutlicht, warum dies der Fall ist.



Abbildung 11: Unavoidable Squares

Betrachtet man in diesem Sudoku beispielsweise die rot markierten Felder, dann fällt auf, dass man durch vertauschen der Dreien mit den Siebenen ebenfalls wieder ein vollständiges Sudoku erhält. Sind diese vier Felder allerdings leer, kann man nicht sagen, in welches Feld, welche Zahl soll: Ergo, das Sudoku hat zwei Lösungen.

Diese Arten von "unavoidable sets" sind nur die einfachsten von sogenannten "unavoidable squares", welche viele möglichen Formen und Ausmaßen annehmen kann, und einen Funktion, welche alle "unavoidable sets" in einem Sudoku findet, ist sehr aufwendig und würde hier den Rahmen sprengen. Deshalb haben wir uns darauf konzentriert, zunächst nur die "unavoidable squares" ausfindig zu machen und dafür zu sorgen, dass mindestens eine von diesen Zahlen in dem Sudoku-Puzzle enthalten ist. So schrieben wir zunächst die folgende Funktion, welche die "unavoidable squares" eines Sudokus in Form einer Liste zurückgibt:

```
def find_unavoidable_squares(sudoku):
    squares = []
    for y in range(8):
        for x in range(8):
            num = sudoku[y,x]
            if num != 0:
                numRight = sudoku[y,x+1]
                for n in range(y+1,9):
                    counterRight = sudoku[n,x]
                    counterNum = sudoku[n,x+1]
                    if counterRight == numRight and counterNum == num:
                        UAS = [(x,y),(x+1,y),(x,n),(x+1,n)]
                        squares.append(UAS)
```

```

        numDown = sudoku[y+1,x]
        for m in range(x+1,9):
            counterDown = sudoku[y,m]
            counterNum = sudoku[y+1,m]
            if counterDown == numDown and counterNum == num:
                UAS = [(x,y),(x,y+1),(m,y),(m,y+1)]
                squares.append(UAS)

    return(squares)

```

Diese Funktion gibt eine Liste von "unavoidable squares" aus und im folgenden ist der Generator, welcher mit dieser Funktion Sudoku-Puzzles erstellt.

```

def make_uas_sudoku(N):
    no_sudoku = True
    while no_sudoku == True:
        sudoku = SudokuGen()
        shuffs = 0
        while shuffs != 25:      #
            pairs = [[x,y] for x in range(9) for y in range(9)]
            squares = find_unavoidable_squares(sudoku)
            shuffle(pairs)
            shuffs = shuffs + 1
            pairs = np.array(pairs)
            runs = 0
            copyshuff = np.copy(sudoku)
            while len(pairs) != N:
                copy = np.copy(copyshuff)
                check_squares = False
                for n in squares:
                    for m in n:
                        if np.array_equal(m,pairs[runs]) == True and len(n) == 1:
                            check_squares = True

                runs = runs + 1
                if check_squares == False:
                    copy[pairs[runs][1],pairs[runs][0]] = 0
                    copyshuff = np.copy(copy)
                    pairs = np.delete(pairs,runs-1,0)
                    runs = 0

                if (runs+1) == len(pairs):
                    break

            if len(pairs) == N:
                sudoku = np.copy(copyshuff)
                break
            if len(pairs) == N:
                no_sudoku = False

    return(sudoku)

```

Der Generator erstellt zunächst ein vollständiges Sudoku, sowie eine vollständige Liste aller Koordinaten der besetzten Felder. Dann werden aus diesem Sudoku die die "unavoidable squares" herausgesucht und in die Liste squares geschrieben. Die Liste der Koordinaten wird gemischt und dann durchgegangen. Ist eine Koordinate der letzte Teil eines "unavoidable squares", so wird diese übersprungen. Alle Koordinaten die nicht übersprungen werden, werden in einer Kopie des Sudokus geleert, bis die gewünschte Anzahl an Hinweisen erreicht ist. Sollte es passieren, dass keine weitere Zahl entfernt werden kann, ohne dass ein "unavoidable squares" vollständig geleert wird, wird die Kopie des Sudokus zurückgesetzt und die Reihenfolge der Koordinaten neu gemischt. Sollte 25-maliges mischen nichts bewirken, wird ein neues Sudoku generiert. Hat aber die Kopie des Sudokus die gewünschte Hinweiszahl erreicht, wird dieses Sudoku ausgegeben.

- Hier kommt noch die Auswertung hin
- 
- 
-

## 4.2 Generator zum erstellen von Sudokus

Nachdem wir unser Verfahren verbessert haben und weitere Verbesserungen( das lokalisieren aller möglichen "unavoidable Sets") hier nicht möglich ist, haben wir versucht einen richtigen Sudoku-Generator zu erstellen. Dieser soll für eine gegebene Anzahl an Hinweisen N, ein eindeutig lösbares Sudoku-Puzzle generieren. Dabei wird im allgemeinen das Verfahren aus Test 5 verwendet, mit dem Unterschied, dass bevor eine Zahl entfernt wird, überprüft wird, ob das entfernen dieser Zahl, dafür sorgt, dass das Sudoku-Puzzle mehrere Lösungen hat. Wenn ja, wird diese Zahl nicht entfernt. So soll die Eindeutigkeit des Sudokus garantiert werden. Sollte der Generator einen Punkt erreichen, an dem er keine Zahl entfernen kann, ohne ein mehrdeutiges Sudoku zu erzeugen, mischt er die Reihenfolge in der er vorgeht neu und versucht es wieder. Sollte 25-maliges neu mischen nicht zum Erfolg führen, wird wie bei Test 5 ein neues vollständiges Sudoku generiert, an welchem das Prozedere wiederholt wird.

Der Code für diesen letzten Generator sieht wie folgt aus:

```
def final_sudoku_generator(N):
    no_sudoku = True
    while no_sudoku == True:
        answer = False
        while answer != True:
            sudoku = np.zeros((9,9))
            sudoku = mkAnswer(sudoku)
            answer = check_sudokuG(sudoku)

            shuffs = 0
            while shuffs != 25:
                pairs = [[x,y] for x in range(9) for y in range(9)]
                squares = find_unavoidable_squares(sudoku)
                shuffle(pairs)
                shuffs = shuffs + 1
                pairs = np.array(pairs)
                runs = 0
                copyshuff = np.copy(sudoku)

                while len(pairs) != N:
                    copy = np.copy(copyshuff)
                    check_squares = False
                    for n in squares:
                        for m in n:
                            if np.array_equal(m,pairs[runs]) == True and len(n) == 1:
                                check_squares = True

                    if check_squares == False:
                        copy[pairs[runs][1],pairs[runs][0]] = 0

                test = empty_sudoku()
                for y in range(1,10):
                    for x in range(1,10):
                        if copy[y-1,x-1] != 0:
                            test = clues(test,int(copy[y-1,x-1]),x,y)

                SUS0 = SudokuSolve(test)
                runs = runs + 1
```

```

if SUSO[0] == 'Eine Lösung' and check_squares == False:
    copyshuff = np.copy(copy)
    pairs = np.delete(pairs, runs-1, 0)
    runs = 0

if SUSO[0] == 'Zwei Lösungen' and (runs+1) == len(pairs):
    break

if SUSO[0] == 'Eine Lösung':
    sudoku = np.copy(copyshuff)
    break

if SUSO[0] == 'Eine Lösung':
    no_sudoku = False

return(sudoku)

```

Wir haben somit einen funktionierenden Generator von Sudokus. In einem letzten Test, wollten wir herausfinden, wie lange unser Generator braucht um bestimmte Sudokus zu generieren. Da es stark vom Zufall abhängt, wie lange der Generator braucht, um ein Sudoku zu generieren, haben wir ihn mit diesem Code immer gleich zehn Sudokus hintereinander generieren lassen:

```

from timeit import default_timer as timer
times = []
hinweise = range(17,81)
hinweise = np.array(hinweise)
hinweise = hinweise[::-1]
for N in hinweise:
    print(N)
    # START MY TIMER
    start = timer()
    for n in range(10):
        final_sudoku_generator(N)
    # STOP MY TIMER
    elapsed_time = timer() - start # in seconds
    print(elapsed_time)
    times.append(elapsed_time)
print(times)

```

Die Ergebnisse finden sich in der folgenden Tabelle:

- Hier kommen die Ergebnisse hin •

-

- Hier kommt Grafik der Ergebnisse hin •
- Abschlussfazit zu generator

## 5 Fazit

Unser Ziel war es, ein funktionierendes Sudoku-Lösungsprogramm zu erstellen. Dies ist uns gelungen. Unser Sudoku-Löser löst eindeutig lösbare Sudokus innerhalb von Bruchteilen einer Sekunde. Allerdings kann er bei Sudokus ohne Lösung und Sudokus mit zwei oder mehr Lösungen etwas länger brauchen. Unser zweites Ziel, auf verschiedene Weisen Sudokus zu generieren und deren Lösbarkeit zu überprüfen haben wir auch erfüllt. Es hat sich herausgestellt, dass eindeutig lösbare Sudokus sehr selten sind und dass die zu bevorzugende Weise sie zu generieren, darin besteht aus einem bereits vollständigen Sudoku, unter Beachtung von "unavoidable sets", Zahlen zu entfernen.

Zuletzt haben wir einen Sudoku-Generator geschrieben, welcher Sudokus mit 23 oder mehr Hinweisen in akzeptabler Zeit erstellen kann. Eindeutig lösbare Sudokus mit weniger Hinweisen sind nämlich sehr selten. Die größte Liste, die wir gefunden haben, von eindeutig lösbare Sudokus mit 17 Hinweisen enthält 50.000 Sudokus, was im Anbetracht der Menge der möglichen Sudokus extrem klein ist.

## 6 Literaturverzeichnis

The Detection of Unavoidable Sets in Sudoku Grids of Different Sizes; Daniel Williams, University of Glamorgan, April 4, 2011

Mathematics of Sudoku I; Bertram Felgenhauer and Frazer Jarvis, February 15, 2006

There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration; Gary McGuire, Bastian Tugemann, Gilles Civario, August 31, 2013

SUDOKU - Strategien zur Lösung; Wolfgang Urban, HIB Wien, 8/2006