Entwicklung des Spiels "Lichter aus"

Jonas Hartmann

Informatik

Prüfer: Herr Grgat

Fachausschussvorsitz: Herr Dietz

Register

Einleitung

Geschichte

Lichter aus ist ein Logik-Puzzlespiel welches in der Form das erste Mal 1995 von Tiger Electronics auf den Markt gebracht wurde.
Das Spiel besteht aus einem 5x5 Kacheln großem Spielfeld. Jede Kachel kann entweder leuchten oder nicht leuchten.

Zu Beginn des Spieles leuchtet eine Kombination an Kacheln. Ziel des Spieles ist alle Kacheln aus zu schalten. Das erreicht man durch drücken der jeweiligen Kacheln. Dabei schaltet eine Kachel jeweils sich selbst, und alle direkt an der Kachel liegenden Nachbarn um. Aus *An* wird *Aus* und aus *Aus* wird *An*

Es gibt verschiedene Lösungsstrategien um das Spiel zu gewinnen. Eine davon nennt sich "Light Chasing".
Dabei arbeitet man sich Zeile für Zeile auf dem Spielfeld nach unten, und schaltet dabei immer, durch das Drücken der Kachel direkt unter einer Leuchtenden, die in der Zeile darüber aus.
Wenn nur Kacheln in der letzten Zeile leuchten gibt es verschiedene Kombinationen die man in der ersten Zeile wieder einschalten muss. Das wird dann wiederholt, bis alle Felder ausgeschaltet sind.

Aufgabenstellung und Zielsetzung

Ziel der Aufgabe war es das Spiel **Lichter Aus** für ein 5x5 Lampen großes Spielfeld in Java oder Python zu programmieren. Ich habe mich für Python entschieden.
Zusätzlich habe ich mir das Ziel gesetzt eine möglichst flexible Version des Spieles zu bauen, die auf jedem quadratischem Spielfeld funktioniert.

Begrifflichkeiten

Um Verwirrungen zu vermeiden sind die Elemente in der Präsentation, dem Spiel und in Code wie folgt definiert:

- **Spielfeld:**
  Die quadratische Anordnung von mehreren Lampen bzw. Kacheln
- **Kachel**
  Die einzelnen Felder des Spielfeldes bzw. die Lampen

Material

Das für die Prüfung abgegebene Material enthält die folgenden Dateien.

```
|– README.md
|– README.pdf
|– debug.py
|– Spiele
    |– einfach
        |– menu.py
        |– game.py
    |– komplex
        |– menu.py
        |– main.py
        |– Solver.py
```

Bei dem Spiel wurde aufgrund der technischer Vorgaben zwischen einer `einfach` und `komplex` Version getrennt.
Diese unterscheiden sich nur in der Weise, wie sie die Spielbretter auf Lösbarkeit untersuchen. Zudem ermöglicht die komplexe Version dem Spieler eine Spielbrettgröße selbst festzulegen. Um die Spiele auszuführen muss die jeweilige `menu.py` gestartet werden.

Bibliotheken

Um die Nutzeroberfläche zu erstellen nutzt das Programm die Bibliothek `Tkinter` diese wird standartmäßig zusammen mit Python installiert.

Die `komplex` Version von dem Programm nutzt zusätzlich noch `Numpy`. Das ist eine Bibliothek für wissenschaftliches Rechnen, um einfach mit Vektoren und Matrizen zu arbeiten. Sie wurde für den komplexen Test auf Lösbarkeit verwendet. Da dort mit Matrizen gerechnet werden muss.

Programm 1 (einfach)

Übersicht

Das Spiel besteht aus zwei Python Dateien. Die `menu.py` und `game.py`. Die erste Datei enthält die *mainloop* für das Spiel. Zudem ist in ihr beschrieben, wie das Menü aufgebaut ist. `game.py` enthält die Beschreibung der LightsOut Klasse. Diese beinhaltet alle Funktionen die für das Spiel selbst benötigt werden.

Das eigentliche Spiel wird durch die LightsOut Klasse abgebildet. Sie enthält zunächst Funktionen, um das Spielfeld zu bauen, Startzustände von diesem zu generieren, die Kacheln umzuschalten.

Die Größe des Spielfeldes wird durch die `GRID_SIZE` Variable in der `game.py` Datei festgelegt. In der einfachen Version ist das nicht zwingend notwendig, da das Feld immer aus 5x5 Kacheln besteht.

Pseudocode

Der Programmcode in der `menu.py` Datei lässt sich vereinfacht wie folgt beschreiben.
`menu.py`

```
def showMenu()
    # clearWindow()
    # Tkinter Grid konfigurieren
    # Textfelder und Buttons erstellen
def startGame()
    # clearWindow()
    # Startet LightsOut Klasse
def clearWindow()
    # Setzt das Programmfenster zurück

mainloop
# Funktion die das Tkinter Fenster startet
```

Die LightsOut Klasse in der `game.py` besitzt die folgende Funktionen.

```
class LightsOut:
    def __init__():
        # Startvariablen (Spielzüge, etc) initialisieren
        self.states = self.generate_states()
        # generate_states() erstellt das Startfeld
        self.build_grid()
        # build_grid() läd das Feld in die Benutzeroberfläche

    def generate_states():
        while True:
            # Zufälliges 5x5 2d-Array generieren lassen
            # Einfacher Test auf Lösbarkeit
            # Wenn die Bedingung erfüllt ist wird die Schleife durchbrochen
            if sum1 % 2 == 0 and sum2 % 2 == 0:
                break
        return states
        # und das Startfeld zurückgegeben

    def build_grid():
        # Vernestete for-Schleifen die über die Koordinaten jeder Kachel
        # iterieren
        for Koordinate in self.states
            # color Variable wird basierend auf dem Zustand
            # der Kachel aus dem states Array
            color = self.get_color(x, y)
            # Für jede Koordinate wird eine Kachel mit Tkinter erstellt
            # Die color Variable wird Füllfarbe der jeweiligen Kachel
            # Anschließend wird die toggle Funktion für den Klick
            # auf die Kachel zugewiesen

    def toggle(x, y):
        # x und y sind die Koordinaten der Kachel auf die gedrückt wurde
        # Jede Nachbarkachel muss daraufhin ihren Zustand umkehren
        # Zusätzlich muss die Nachbarkachel innerhalb
        # des Spielfeldes liegen.
        # Spielzugzähler um 1 erhöhen
        # und Label aktualisieren
        # Zusätzlich muss geprüft werden ob das Spiel gewonnen wurde
```

```
    def get_color(x, y):
        # Gibt die Farbe des jeweiligen Spielfeldes zurück

    def check_win():
        # Überprüft ob



    def show_win_message():
        # Öffnet ein neues Fenster
        # mit der Anzahl an benötigten Zügen

    def reset():
        # Setzt das Spielfeld zurück und
        # erstellt ein neues

    def showGame():
        # Wird bei der Initialisierung der Klasse aufgerufen
        # um die Benutzeroberfläche von dem Menü
        # zum Spiel zu verändern
```

Spielfeld

Um das Spielfeld für das Programm verständlich zu machen werden die Zustände der Lampen (Kacheln) durch 0 und 1 abgebildet.
Eine 1 repräsentiert hierbei den Zustand "Ein" von der Lampe bzw. Kachel.
Eine 0 stellt dann folglich den Zustand "Aus" dar.

Basierend darauf wird zufällig ein quadratisches Spielfeld mit der Seitenlänge 5 generiert.

Das Spielfeld wird in `generate_states` Funktion der LightsOut Klasse erstellt.

```
states =
[[random.choice([0, 1]) for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]
```

`random.choice([0, 1])` füllt die jeweilige Position zufällig mit einer 0 oder einer 1.

Daraus ergibt sich dann ein 2-dimensionalem Array welcher wie folgt aussieht:

```
states  =  [[1,0,0,0,0],
            [0,1,0,1,0],
            [0,0,1,0,0],
            [0,0,1,1,1],
            [1,0,0,0,0]]
```

Validierung des Spielfeldes

Da nicht jede generierte Kombination von Zuständen auf dem 5x5 Feld lösbar ist, muss diese erst validiert werden.
Deswegen muss das zufällig generierte Feld erst auf seine Lösbarkeit überprüft werden. Für diese Überprüfung kann bei einem 5x5 Feld folgender Ansatz verwendet werden.

```
# Schritt 1
[1, 0, 0, 1, 0]    [1, 0, ., 1, 0]
[1, 1, 1, 1, 0]    [., ., ., ., .]
[1, 1, 1, 0, 0] => [1, 1, ., 0, 0]
[0, 0, 1, 0, 0]    [., ., ., ., .]
[1, 1, 1, 1, 1]    [1, 1, ., 1, 1]

# Schritt 2
[1, 0, 0, 1, 0]    [1, ., 0, ., 0]
[1, 1, 1, 1, 0]    [1, ., 1, ., 0]
[1, 1, 1, 0, 0] => [., ., ., ., .]
[0, 0, 1, 0, 0]    [0, ., 1, ., 0]
[1, 1, 1, 1, 1]    [1, ., 1, ., 1]
```

Im ersten Schritt muss in die erste, die dritte und die fünfte Zeile des Spielfeldes geschaut werden. Dort wählt man jeweils die ersten beiden und die letzten beiden Werte aus. Addiert müssen diese dann eine gerade Zahl ergeben.

Der zweite Schritt ähnelt dem ersten Schritt. Allerdings betrachtet man dort die ersten beiden und die letzten beiden Zeilen und aus diesen jeweils den ersten, dritten und fünften Wert. Auch diese müssen in Summe eine gerade Zahl ergeben.

Dieser Test wurde von Jaap Scherphuis übernommen
https://puzzling.stackexchange.com/a/123076 (01.06.2025)

Im Code ist dieser Test in der `generate_states` Funktion implementiert.

Nachdem die Startzustände zufällig generiert wurden werden mit den Variablen `testSum1` und `testSum2` die Summen für den Test gebildet.

```
# Auswahl der relevanten Felder aus der Startmatrix
testSum1 = self.states[0][0]+self.states[0][1]+...+self.states[4][4]
testSum2 = self.states[0][0]+self.states[0][2]+...+self.states[4][4]
```

(Durch die Punkte wurden die Summen zur besseren Lesbarkeit gekürzt)

`self.states` ist der zweidimensionale Array der die Startkonfiguration des Spielfeldes enthält. Durch `self.states[0][0]` wird der erste Wert mit der Koordinate (1|1) vom Spielfeld geladen, da der erste Wert in einem Array den Index 0 und nicht 1 hat.

Der gesamte Prozess spielt sich in einer While-Schleife innerhalb der `generate_states` Funktion ab. Nach jedem Durchlauf wird geprüft, ob die Summen 1 und 2 gerade sind.

```
# Wenn beide Summen gerade sind
if sum1 % 2 == 0 and sum2 % 2 == 0:
    # Dann wird die While-Schleife durchbrochen
    break
```

Sobald die Bedingung erfüllt ist, wird die While-Schleife geschlossen und der 2-dimensionale Array wird zurückgegeben.

Spielfeld als graphische Oberfläche (GUI)

Das Spiel benötigt jetzt eine Benutzeroberfläche die dieses Spielfeld abbildet. Dafür besitzt die LightsOut Klasse die `build_grid` Funktion.
Sie besteht aus zwei in einander liegenden For-Schleifen die je über die Größe der `GRID_SIZE` iterieren.

Für jede Kachel legen sie die `color` Variable fest.

```
# Wenn Zustand von Kachel 1, dann color = 'yellow'
color = 'yellow' if self.states[y][x] == 1 else 'black'
```

Wenn der Zustand der Kachel gleich 1 ist, so wird die Variable auf den String 'yellow' gesetzt. Wenn nicht dann auf 'black'. Dieser Prozess wiederholt sich, bis jede Kachel ihre Farbe bekommen hat.

Damit diese dann auch auf dem Spielfeld in der Benutzeroberfläche sichtbar ist muss in Tkinter noch ein Rechteck für jede Kachel angelegt werden.
Das geschiet wie folgt:

```
self.game.create_rectangle(x * FIELD_SIZE, y * FIELD_SIZE, (x + 1) * FIELD_SIZE, (y + 1) *
```

Als Füllfarbe wird die `color` Variable angegeben. `FIELD_SIZE` beschreibt die Breite der Kacheln in Pixeln. Diese ist auch vordefiniert.
Die ersten beiden Parameter sind die x und y Startkoordinate für das Rechteck. Die zweiten beiden Parameter sind die Endkoordinaten.

Umschalten der Felder

Eine Kachel kann sich, wie bereits beschrieben, in einem von zwei Zuständen befinden. Diese sind als 0 (Lampe aus) und 1 (Lampe an) definiert.

Die Zustandsänderung wird von der toggle Funktion ausgelöst. Diese Funktion benötigt eine x- und eine y-Koordinate als Parameter. Basierend auf dieser Koordinate berechnet die Funktion dann die Nachbarkacheln.

```
for dx, dy in [(0, 0), (-1, 0), (1, 0), (0, -1), (0, 1)]:
    nx, ny = x + dx, y + dy
    # Feld muss innerhalb des Rasters liegen
    if 0 <= nx < GRID_SIZE and 0 <= ny < GRID_SIZE:
        self.states[ny][nx] ^= 1
```

In diesem Auszug aus der toggle Funktion passiert vieles auf einmal. Zuerst legt die for-Schleife die Variablen nx und ny fest. Diese beschreiben das Delta x und das Delta y der Nachbarkacheln relativ zu der Kachel welche die toggle Funktion ausgelöst hat.

Im nächsten Schritt wird dann die Koordinate des jeweiligen Nachbarn berechnet. Diese sind durch nx und ny beschrieben und errechnen sich aus der Summe von x und dx, sowie y und dy.

Für jeden errechneten Nachbarn gibt es dann noch eine Kontrolle, ob die Kachel auf dem Spielfeld liegt. Die jeweilige x- und y-Koordinate müssen zwischen 0 und der Spielfeldbreite `GRID_SIZE` liegen.

Das wiederholt sich dann bis alle Nachbarn die oben in der Liste in der for-Schleife definiert wurden bestimmt sind.

Wenn die Kachel auf dem Spielfeld liegt wird auf den dazugehörigen Eintrag in dem states Array, welcher die Zustände der Felder speichert umgeschaltet. 0 wird zu 1 und 1 wird zu 0. Das passiert mit dem sogenannten **XOR-Operatior** (Exklusiv-Oder-Gatter). In Python wird dieser mit folgender Syntax verwendet `^=`.

Die XOR Rechung funktioniert wie folgt:
Wenn der **Zustand 0** ist und die **XOR Operation mit 1** ausgeführt wird, dann wird der **Zustand zu 1**.
Wenn der **Zustand 1** ist und die **XOR Operation mit 1** ausgeführt wird, dann wird der **Zustand zu 0**.

Tablellarische Darstellung:

Zustand	XOR	Ergebnis
1	1	0
0	1	1

Beispiel:

```
toggle(x=3, y=2):
    for dx, dy in [(0, 0), (-1, 0), (1, 0), (0, -1), (0, 1)]:
        nx, ny = x + dx, y + dy
        # Feld muss innerhalb des Rasters liegen
        if 0 <= nx < GRID_SIZE and 0 <= ny < GRID_SIZE:
            self.states[ny][nx] ^= 1
```

Im ersten Durchgang der for-Schleife ist `dx=0` und `dy=0`. Somit ergibt `nx = 3 + 0 = 3` und `ny = 2 + 0 = 2`. Das entspricht dem Feld mit der Koordinate (3|2).

Im zweiten Durchgang ist `dx=-1` und `dy=0`. Damit ergibt sich für `nx = 3 - 1 = 2`. Das ist das Feld (2|2), und somit der linke Nachbar.

Das ganze wiederholt sich dann für alle dx und dy Werte.

Programm 2 (komplex)

Mit dem ersten Programm konnte man auf 5x5 Felder großen Spielfelder spielen. Ziel war es aber ein Programm zu schreiben, welches auch mit anderen Spielfelder arbeiten kann.

Die `toggle` Funktion ist nicht an die Größe des Spielfeldes gebunden. Sie arbeitet mit der `GRID_SIZE` Variable, die bei dem Start des Programmes auf `5` gesetzt wurde.

Das einzige Problem bereitet bei der Skalierung also nur der Test auf Lösbarkeit der Startkonfiguration.

Es gibt aber eine mathematische Möglichkeit ein beliebig großes Feld auf Lösbarkeit zu untersuchen. Diese ist zwar komplexer, dafür aber flexibel in der Feldgröße.

Der komplexe Test auf Lösbarkeit

Ansatz

Die Reihenfolge in der die Felder geschaltet werden ist irrelevant. Desweiteren muss man für die "optimalste" Lösung jedes Feld nur maximal ein Mal umschalten, da es bei zwei Mal umschalten wieder in seinem Ausgangszustand wäre.

Mit dieser Annahme können wir die Startkonfiguration als einen Vektor $\overrightarrow{a}$ darstellen.
Dieser Vektor besteht aus den Startzuständen der Kacheln, die Zeile für Zeile untereinander geschrieben werden.

Des Platzes halber werde ich diese Rechnung mit einem 2x2 Spielbrett zeigen. Die Rechnung ist aber für jedes quadratische Feld möglich.

Startkonfiguration:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Daraus ergibt sich der Vektor $\overrightarrow{a}$

$$\overrightarrow{a} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Als nächstes muss eine Übergangsmatrix M definiert werden. Diese Matrix beschreibt welche Lampe welche Lampen schaltet.

Für ein 2x2 Spielbrett sieht diese so aus.

$$M = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Da wir wissen, dass es Ziel des Spiels ist, alle Lampen auszuschalten, können wir einen Ziel- / Ergebnis-Vektor $\overrightarrow{b}$ aufstellen. Dieser stellt das Spielbrett mit allen Lampen aus dar.

$$\overrightarrow{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Zuletzt definieren wir noch einen Schalt-Vektor $\overrightarrow{x}$. Dieser ist die uns unbekannte Kombination an Feldern, die geschaltet werden müssen, um in den Zielzustand (Ergebnis-Vektor) zu gelangen.

$$\overrightarrow{x} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

Dank linearer Algebra wissen wir, dass:

$$M\overrightarrow{x} + \overrightarrow{a} = \overrightarrow{b}$$

Nach $\overrightarrow{x}$ umgestellt ist das:

$$\overrightarrow{x} = \frac{(\overrightarrow{b} - \overrightarrow{a})}{M}$$

Statt durch die Matrix M zu dividieren kann auch mit M^{-1} multipliziert werden.

$$\vec{x} = M^{-1}(\vec{b} - \vec{a})$$

Somit ergibt sich für unser 2x2 Spielbrett folgende Lösung:

$$\begin{bmatrix}0\\1\\1\\0\end{bmatrix} = \begin{bmatrix}1&1&1&0\\1&1&0&1\\1&0&1&1\\0&1&1&1\end{bmatrix}^{-1} \left(\begin{bmatrix}0\\0\\0\\0\end{bmatrix} - \begin{bmatrix}0\\1\\1\\0\end{bmatrix} \right)$$

Das heißt, um das als M definierte Spielbrett zu lösen muss man Feld 2 und Feld 3 umschalten.

Modulare Arithmetik

Die modulare Arithmetik kennen wir zum Beispiel von der 12- und 24-Stunden Uhr. In der Programmierung haben wir sie bereits durch den Modulo Operator kennengelernt. Dieser hat die Funktion eine Zahl durch eine andere zu teilen und den Rest auszugeben. In der Praxis mit der Uhrzeit sieht das so aus:

```
# 24h Zeit in 12h Zeit umwandeln
14 % 12 = 2 # <- Da 14/12 = 1 Rest 2 14 Uhr = 2 Uhr
16 % 12 = 4 # <- Da 16/12 = 1 Rest 4 16 Uhr = 4 Uhr
```

Dieses System lässt sich für das Spiel `Lichter aus` anwenden, da wir uns dort in Modulo 2 befinden. Es gibt nur die Zustände 0 und 1. Aus einer 4 würde wieder eine 0 werden. Das geht, da es nur zwei Zustände gibt, in denen die Felder sein können.
Wenn ein Feld 4 mal geschaltet wird ist es im gleichen Zustand wie wenn es kein Mal geschaltet wurde.

Implementierung

Implementiert wurde der Test auf Lösbarkeit in der Klasse `Solver`.

Test auf Lösbarkeit

Zunächst wird die Übergangsmatrix für die vorgegebene Feldgröße in der `build_matrix()` Funktion erstellt.

```
def build_matrix(self):
    # Übergangsmatrix (Welche Lampe schaltet welche Lampen) erstellen
    A = np.zeros((self.n, self.n), dtype=int)
    for i in range(self.n):
        A[i][i] = 1
        row, col = divmod(i, self.size)
        for r, c in [(row-1, col), (row+1, col), (row, col-1), (row, col+1)]:
            # Test ob Kachel innerhalb des Spielfeldes ist
            if 0 <= r < self.size and 0 <= c < self.size:
                A[i][r * self.size + c] = 1
    return A
```

A wird als Matrix, bestehend aus 0en mit der Feldgröße n*n. Im nächsten Schritt wird dann mit einer For-Schleife über die einzelnen Zeilen der Matrix iteriert. Die zweite For-Schleife läuft dann über alle Nachbarn und setzt diese auf den Zustand 1.

Im nächsten Schritt wird die Inverse im Modulo 2 Zahlenraum der Übergangsmatrix mithilfe des Gausschen Elimination ermittelt. Das passiert in der `get_solutionMatrix` Funktion der Solver Klasse.

Diese Funktion gibt auch schon die optimale Lösung für das gegebene Spielfeld zurück.

Um zu überprüfen, ob das Feld mit der gegebenen Lösung lösbar ist werden die Werte wieder in die Gleichung eingesetzt.

$$M\vec{x} + \vec{a} = \vec{b}$$

```
testMatrix =
np.add(np.linalg.matmul(self.transitionMatrix, solutionMatrix), stateVector)
```

`np.add` addiert zwei Matrizen, `np.linalg.matmul` ist eine Funktion von Numpy um die beiden Faktoren miteinander zu multiplizieren.

Danach wird überprüft ob die errechnete Lösung nur aus 0en besteht, dann ist das Feld lösbar und die Gleichung erfüllt. Zusätzlich wird geprüft, dass die Startkonfiguration nicht nur als 0en besteht, denn dann wäre das Feld schon gelöst.

```
# Ist true, wenn die errechnete Lösung nur aus 0en besteht
# Wenn true, dann ist das Feld lösbar
allZeros = not np.any(testMatrix)

# Ist true, wenn die Startkonfiguration nicht nur 0en enthält
# Dann wäre das Spielbrett bereits gelöst
startMatrix = np.any(stateMatrix)

# Wenn beide Bedingungen erfüllt sind, dann ist das Spielbrett lösbar
if allZeros and startMatrix:
    isSolvable = True
```

Da die Lösungsmatrix als Vektor angegeben ist muss dieser wieder in eine n*n große Matrix umgewandelt werden. Das passiert mit der Numpy reshape Funktion.

```
solutionMatrix = np.reshape(solutionMatrix, (self.size, self.size))
```

Am Ende gibt die gesamte Solve Funktion der Solver Klasse einen Array zurück der aus der Lösungsmatrix, der Lösbarkeit sowie der vergangenen Zeit besteht.

```
return[solutionMatrix, isSolvable, timeEnd - timeStart]
```

Das Spiel fährt dann mit der `bulid_grid` Funktion fort.

Solve Funktion

Auch neu in der komplexen Version ist die `solve` Funktion im Spiel. Diese löst das Spielfeld automatisch mit einer Animation.

Da die optimale Lösung bereits durch den Test auf Lösbarkeit bekannt ist, müssen auf dem Spielfeld dann nur noch die Kacheln umgeschaltet werden, für die der Solver in der Lösungsmatrix *1* zurückgegeben hat.

Diese `solve` Funktion funktioniert wie folgt:

```
def solve(self):
    # Greift die Lösungsmartix aus dem Solver ab
    solutionMatrix = self.solver.solve(self.states)[0]
    # For-Schleifen über die Kacheln in der Lösungsmatrix
    for y in range(GRID_SIZE):
        for x in range(GRID_SIZE):
            # Wenn diese Kachel den Wert 1 hat
            if solutionMatrix[y][x] == 1:
                # Dann schalte diese um
                self.toggle(x, y)
                # Kurze Pause, damit es ersichtlich ist,
                # welches Feld geschaltet wurde
                time.sleep(1/GRID_SIZE)
                # Nach jedem Schalten muss die GUI
                # (self.root) aktualisiert werden
                self.root.update()
```

`time.sleep(1/GRID_SIZE)` wurde gewählt, damit die `solve` Funktion auf größeren Feldern schneller läuft und nicht ewig braucht.

Weiterentwickling

Alternative Spielfeld Erstellung

Anstatt die Spielfelder komplett zufällig zu erstellen hätte man auch mit einem lösbaren, bzw. gelösten Feld starten können.
Um dann die Startkonfiguration zu erstellen müssten zufällig gewählte Felder umgeschaltet werden, damit das Spielfeld in einem nicht bereits gelösten Zustand ist.

Highscore

Ein Scoring System in dieser Version des Spieles einzubauen ist nicht sinnvoll, da jedes Feld komplett zufällig generiert wird und somit immer eine unterschiedliche Anzahl an Zügen zum lösen benötigt werden.
Würde man aber verschiedene Konfigurationen, die für jeden gleich sind, als Level vordefinieren, könnte man ein Leaderboard für die Spieler erstellen und in einer Datenbank bzw. einer Datei speichern.

Zusätzliche Funktionen

Zusätzlich könnte man Spieler selbst Spielfelder entwerfen lassen, die sie dann lösen könnten. Diese müsste man dann beispielsweise in einer JSON Datei oder einer Datenbank speichern. Das würde den Rahmen der Prüfung wirklich sprengen.

Quellen

Grundlagen

https://en.wikipedia.org/wiki/Lights_Out_(game) (19.04.2025)
Geschichte von Lights out

https://www.youtube.com/watch?v=kTnYls94-r8 (19.04.2025)
'Lights out' Game (Light Chasing)

https://mathworld.wolfram.com/LightsOutPuzzle.html (20.04.2025)

https://mathematikalpha.de/logikspiel-licht-aus (19.04.2025)

Einfacher Test

https://puzzling.stackexchange.com/questions/123075/how-do-i-determine-whether-a-5x5-lights-out-puzzle-is-solvable-without-trying-to (21.04.2025)
https://puzzling.stackexchange.com/q/123075 (21.04.2025)
Einfacher Test auf Lösbarkeit des 5x5 Feldes

https://www.jaapsch.net/puzzles/lomath.htm#solvtest (21.04.2025)
Zugehörig zum einfachen Test von 5x5 Feldern

Komplexer Test

https://www.youtube.com/watch?v=0fHkKcy0x_U (20.04.2025)
Zum Testen der Lösbarkeit des Startfeldes.

https://de.wikipedia.org/wiki/Modulare_Arithmetik (24.04.2025)
Modulare Arithmetik (Beispiel Uhrzeit)

Programmierung

https://de.wikipedia.org/wiki/NumPy (01.06.2025)
Numpy Bibliothek

https://numpy.org/doc/stable/reference/generated/numpy.add.html (21.04.2025)

https://numpy.org/doc/stable//reference/generated/numpy.linalg.matmul.html (21.04.2025)

https://numpy.org/doc/stable/reference/generated/numpy.average.html (21.04.2025)

https://www.geeksforgeeks.org/how-to-create-full-screen-window-in-tkinter/ (01.06.2025)
Tkinter-Fullscreen Fenster

Mathematik

https://www.jaapsch.net/puzzles/lomath.htm (21.04.2025)

https://de.wikipedia.org/wiki/Exklusiv-Oder-Gatter#Symbolik (22.04.2025)
Exklusiv-Oder-Gatter (XOR) für toggle Funktion

https://www.jaapsch.net/puzzles/lomath.htm#inverse (22.04.2025)
https://de.wikipedia.org/wiki/Pseudoinverse (22.04.2025)

Quellcode

Programm 1 `menu.py`

```
import tkinter as tk
from game import LightsOut

# Rendert Hauptmenü
def showMenu():
    clearWindow()
    root.title("Lichter Aus - Hauptmenü")
    root.grid()

    # Jede Spalte im Grid auf die gleiche Breite setzen
    for i in range(3):
        root.grid_columnconfigure(i, weight=1)

    # Label für den Titel
    title_label = tk.Label(root, text="Lichter Aus", font=("Arial", 32))
    title_label.grid(row=0, column=1, pady=200)

    # Button zum Starten des Spiels
    start_button = tk.Button(root, text="Spiel starten",
                             command=lambda:startGame())
    start_button.grid(row=1, column=1, pady=10)

    # Button zum Beenden des Spiels
    exit_button = tk.Button(root, text="Beenden", command=root.quit)
    exit_button.grid(row=2, column=1, pady=10)

def startGame():
    clearWindow()
    # Neue Spielinstanz starten
    LightsOut(root, showMenu)

def clearWindow():
    # Alle Widgets im Fenster entfernen
    for widget in root.winfo_children():
        widget.destroy()


if __name__ == "__main__":
    root = tk.Tk()
```

```
    root.geometry("500x680")
    root.attributes('-fullscreen',True)

    showMenu()

    root.mainloop()
```

Programm 1 `game.py`

```
import tkinter as tk
import random

GRID_SIZE = 5  # Könte dynamisch angepasst werden
# (Fensterbreite - Padding vom Spielfeld) / Größe des Grids (5 Felder)
FIELD_SIZE = (500-20)/GRID_SIZE  # Größe der Felder in Pixeln


class LightsOut:
    def __init__(self, root, backToMenu):
        # Hauptfenster übergeben
        self.root = root

        # Startvariablen definieren
        self.move_count = 0
        self.round_count = 0

        # Spielfenster erstellen
        self.showGame(backToMenu)

        # Zustände des Spiels initialisieren
        self.fields = {}
        self.states = self.generate_states()

        # Spielfeld aufbauen
        self.build_grid()

    def generate_states(self):
        print("Generiere Zustände")
        # Run counter der die Durchläufe zählt,
        # bis eine lösbare Startkonfiguration gefunden wurde
        run = 1

        # While-Schleife läuft solange, bis eine lösbare Konfiguration
        # gefunden wurde
        while True:
            # Zufällige Konfiguration bestehend aus 0 und 1 generieren
            states = [[random.choice([0, 1]) for _ in range(GRID_SIZE)]
                      for _ in range(GRID_SIZE)]
```

```
            # Einfacher Test auf Lösbarkeit durch Summen
            # https://puzzling.stackexchange.com/a/123076
            sum1 = states[0][0]+states[0][1]+states[0][3]+states[0][4]
                    +states[2][0]+states[2][1]+states[2][3]+states[2][4]
                    +states[4][0]+states[4][1]+states[4][3]+states[4][4]
            sum2 = states[0][0]+states[0][2]+states[0][4]+states[1][0]
                    +states[1][2]+states[1][4]+states[3][0]+states[3][2]
                    +states[3][4]+states[4][0]+states[4][2]+states[4][4]

            # Run counter mit jedem Durchlauf erhöhen
            print(f"Try: {run}")
            run += 1

            # Wenn beide Summen gerade sind, also die Konfiguration lösbar ist
            if sum1 % 2 == 0 and sum2 % 2 == 0:
                # Dann wird die While-Schleife durchbrochen und
                # anschließend die Konfiguration zurückgegeben
                break
        return states

    def build_grid(self):
        # Button für Kacheln auf dem Spielfeld erstellen
        # Beide Schleifen iterieren über die x und y Koordinaten der Kacheln
        for y in range(GRID_SIZE):
            for x in range(GRID_SIZE):
                # Wenn der Zustand des jeweiligen Kachel 1 (an) ist,
                # dann wird das Feld gelb gefärbt wenn nicht dann schwarz
                color = self.get_color(x, y)
                # Field beschreibt die Rechtecke (Kacheln) auf dem Canvas
                # Diese sind werden mit color, wie oben für das Kachel
                # definiert ausgefüllt
                field = self.game.create_rectangle(x * FIELD_SIZE,
                            y * FIELD_SIZE, (x + 1) * FIELD_SIZE,
                            (y + 1) * FIELD_SIZE, fill=color,
                            outline='black')
                self.game.grid_configure(row=0, column=0, columnspan=GRID_SIZE)
                # self.fields ist ein Dictionary, welches den Koordinaten
                # den Kacheln Nummern (IDs) zuordnet
                self.fields[(x, y)] = field
```

```
            # Diese ID wird benötigt um den Kacheln (canvas rectangles)
            # Events zuzuordnen
            # Das Event bedeutet hier, dass ein Klick auf das Feld
            # self.toggle(x, y) aufruft
            self.game.tag_bind(field, '<Button-1>', lambda event, x=x, y=y:
                               self.toggle(x, y))

    # Toggle-Funktion um die Kacheln umschalten
    def toggle(self, x, y):
        # Beim Klicken die angrenzenden Kacheln umschalten
        # For-Schleife iteriert über die angrenzenden Kacheln
        for dx, dy in [(0, 0), (-1, 0), (1, 0), (0, -1), (0, 1)]:
            # nx und ny sind die Koordinaten der jeweiligen Nachbarkacheln
            nx, ny = x + dx, y + dy
            # Das Feld muss innerhalb des Rasters liegen
            # Es wird nur umgeschaltet, wenn das Feld innerhalb des
            # Spielfelds liegt
            if 0 <= nx < GRID_SIZE and 0 <= ny < GRID_SIZE:
                # XOR-Operator wird als Umschalter verwendet
                self.states[ny][nx] ^= 1
                self.game.itemconfig(self.fields[(nx, ny)],
                                     fill=self.get_color(nx, ny))

        # Zähler erhöhen (+= 1) und Label aktualisieren
        self.move_count += 1
        self.clicks_label.config(text=f"Züge: {self.move_count}")

        # Nach jedem Zug prüfen, ob das Spiel gewonnen wurde
        if self.check_win():
            self.show_win_message()

    # Gibt die Farbe der Kachel x y zurück
    def get_color(self, x, y):
        # Wenn der Zustand des Feldes 1 ist yellow, sonst black
        return "yellow" if self.states[y][x] == 1 else "black"

    # Prüft ob das Spiel gewonnen wurde
    def check_win(self):
        # Prüfen, ob das Spiel gewonnen wurde (alle Felder sind schwarz)
        return all(not cell for row in self.states for cell in row)
```

```
    # Popup-Fenster für Gewinnnachricht
    def show_win_message(self):
        # Kreeiert ein neues Fenster, mit dem Titel "Gewonnen!"
        win_popup = tk.Toplevel(self.root)
        win_popup.title("Gewonnen!")
        # Kurze Nachricht, in wie vielen Zügen Spiel gewonnen wurde
        tk.Label(win_popup, text=f"Du hast in {self.move_count} Zügen gewonnen!",
                font=("Arial", 14)).pack(pady=10)
        # Rundenanzahl um 1 erhöhen
        self.round_count += 1
        self.rounds_label.config(text=f"Runde: {self.round_count}")
        # Lambda um Popup zu schließen und gleichzeitig das Spiel zurückzusetzen
        # Lambda, da mehrere Funktionen bei einem Klick ausgeführt werden sollen
        tk.Button(win_popup, text="OK", command=lambda:
                        [self.reset(), win_popup.destroy()]).pack(pady=5)

    # Reset-Funktion für das Spielfeld, generiert neue Startkonfiguration
    def reset(self):
        print("Zurücksetzen")
        # Züge zurücksetzen
        self.move_count = 0
        # Fields leeren
        self.fields = {}
        # Label aktualisieren
        self.clicks_label.config(text="Züge: 0")

        # Neu generierte Zustände für die jeweiligen Felder
        self.states = self.generate_states()
        # Baut das neue Spielfeld mit den neuen Zuständen auf
        self.build_grid()

    # Canvas für Spiel erstellen
    def showGame(self, backToMenu):
        # Ändert den Titel des Fensters
        self.root.title("Lichter Aus - Spiel")
        # Erstellt das Spielfeld als Canvas
        # Breite und Höhe werden aus Anzahl mal der Größe der Felder berechnet
        self.game = tk.Canvas(self.root, width=GRID_SIZE * FIELD_SIZE,
        height=GRID_SIZE * FIELD_SIZE)
```

```
        # Fügt das Canvas Element dem Grid hinzu
        self.game.grid(padx=10, pady=10)

        self.clicks_label = tk.Label(self.root, text="Züge: 0",
        font=("Arial", 14))
        self.clicks_label.grid(row=GRID_SIZE, column=0, columnspan=GRID_SIZE,
        pady=10)

        self.rounds_label = tk.Label(self.root, text="Runde: 0",
        font=("Arial", 14))
        self.rounds_label.grid(row=GRID_SIZE + 1, column=0, columnspan=GRID_SIZE,
        pady=10)

        self.solve_button = tk.Button(self.root, text="Menü", command=backToMenu)
        self.solve_button.grid(row=GRID_SIZE + 2, column=0, columnspan=GRID_SIZE,
        pady=10)

        self.reset_button = tk.Button(self.root, text="Zurücksetzen",
        command=self.reset)
        self.reset_button.grid(row=GRID_SIZE + 3, column=0, columnspan=GRID_SIZE,
        pady=10)
```

Programm 2 `menu.py`

```
import tkinter as tk
from game import LightsOut

# Rendert Hauptmenü
def showMenu():
    clearWindow()
    root.title("Lichter Aus – Hauptmenü")
    root.grid()

    for i in range(3):
        root.grid_columnconfigure(i, weight=1)

    # Label für den Titel
    title_label = tk.Label(root, text="Lichter Aus", font=("Arial", 32))
    title_label.grid(row=0, column=1, pady=80)

    # Label für die Feldgröße
    size_label = tk.Label(root, text="Feldgröße:", font=("Arial", 12))
    size_label.grid(row=1, column=1, pady=10)

    # Eingabefeld für die Feldgröße
    size_entry = tk.IntVar(value = 2)
    tk.Spinbox(root, from_=1, to=100, increment=1, width=5,
    textvariable=size_entry).grid(row=2, column=1, pady=10)
    # Button zum Starten des Spiels
    start_button = tk.Button(root, text="Spiel starten",
    command=lambda:startGame(size_entry.get()))
    start_button.grid(row=3, column=1, pady=10)

    # Button zum Beenden des Spiels
    exit_button = tk.Button(root, text="Beenden", command=root.quit)
    exit_button.grid(row=4, column=1, pady=10)

def startGame(size_entry):
    # Größe des Spielfelds aus dem Eingabefeld holen
    gridSize = size_entry
    # Fenster zurücksetzen
    clearWindow()
    # Neue Spielinstanz starten
```

```
    LightsOut(root, gridSize, showMenu)

def clearWindow():
    # Alle Widgets im Fenster entfernen
    for widget in root.winfo_children():
        widget.destroy()


if __name__ == "__main__":
    root = tk.Tk()
    root.geometry("500x750")
    #root.attributes('-fullscreen',True)

    showMenu()

    root.mainloop()
```

Programm 2 `game.py`

```
import tkinter as tk
import numpy as np
import time
import random
from Solver import Solver


class LightsOut():
    def __init__(self, root, gridSize, backToMenu):
        # GRID_SIZE und FIELD_SIZE als globale Variablen definieren
        # Damit andere Funktionen auf die Werte zugreifen können
        global GRID_SIZE
        global FIELD_SIZE

        GRID_SIZE = gridSize  # Größe des Spielfelds
        FIELD_SIZE = (500-20)/GRID_SIZE  # Größe der Felder in Pixeln
        print(FIELD_SIZE)

        self.root = root
        self.root.title("Lichter Aus - Spiel")

        self.solver = Solver(size=GRID_SIZE)

        self.game = tk.Canvas(self.root, width=GRID_SIZE * FIELD_SIZE,
        height=GRID_SIZE * FIELD_SIZE)
        self.game.grid(padx=10, pady=10)

        self.fields = {}
        self.generate_states()

        self.build_grid()

        self.move_count = 0
        self.round_count = 1

        self.clicks_label = tk.Label(self.root, text="Züge: 0",
        font=("Arial", 14))
        self.clicks_label.grid(row=1, column=0, columnspan=GRID_SIZE, pady=10)
```

```
        self.rounds_label = tk.Label(self.root,
        text=f"Runde: {self.round_count}", font=("Arial", 14))
        self.rounds_label.grid(row=2, column=0, columnspan=GRID_SIZE, pady=10)

        self.solve_button = tk.Button(self.root, text="Menü",
        command=backToMenu)
        self.solve_button.grid(row=3, column=0, columnspan=GRID_SIZE, pady=10)

        self.solve_button = tk.Button(self.root, text="Lösen",
        command=self.solve)
        self.solve_button.grid(row=4, column=0, columnspan=GRID_SIZE, pady=10)

        self.reset_button = tk.Button(self.root, text="Zurücksetzen",
        command=self.reset)
        self.reset_button.grid(row=5, column=0, columnspan=GRID_SIZE, pady=10)

    def generate_states(self):
        print("Generiere Zustände")
        # Run counter der die Durchläufe zählt, bis eine lösbare
        # Startkonfiguration gefunden wurde
        run = 1
        # While-Schleife läuft solange, bis eine lösbare Konfiguration
        # gefunden wurde
        while True:
            # Zufällige Konfiguration bestehend aus 0 und 1 generieren
            self.states = np.array([[random.choice([0, 1])
                        for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)])
            # Solver mit der Startkonfiguration aufrufen, gibt die
            # Lösbarkeit und eine mögliche Lösung zurück
            solved = self.solver.solve(self.states)
            print(f"Try: {run}, {solved[1]}, Time: {solved[2]}")
            # Run counter mit jedem Durchlauf erhöhen
            run += 1
            # Wenn der Solver true zurückgibt, also die Konfiguration
            # lösbar ist
            # Dann wird die While-Schleife durchbrochen und anschließend
            # die Konfiguration zurückgegeben
            if solved[1] == True:
                break
        return self.states
```

```
def build_grid(self):
    # Button auf dem Canvas erstellen
    # Beide Schleifen iterieren über die x und y Koordinaten der
    # Felder
    for y in range(GRID_SIZE):
        for x in range(GRID_SIZE):
            # Wenn der Zustand des jeweiligen Feldes 1 (an) ist,
            # dann wird das Feld gelb gefärbt wenn nicht dann schwarz
            color = 'yellow' if self.states[y][x] == 1 else 'black'
            # Field beschreibt die Rechtecke (Kacheln) auf dem
            # Spielfeld (game)
            # Diese sind werden mit color, wie oben für das Feld
            # definiert ausgefüllt
            field = self.game.create_rectangle(x * FIELD_SIZE,
            y * FIELD_SIZE, (x + 1) * FIELD_SIZE, (y + 1) * FIELD_SIZE,
            fill=color, outline='black')
            self.game.grid_configure(row=0, column=0,
            columnspan=GRID_SIZE)
            # self.fields ist ein Dictionary, welches den Koordinaten
            # der Kacheln Nummern (IDs) zuordnet
            self.fields[(x, y)] = field
            # Diese ID wird benötigt um den Kacheln (canvas rectangles)
            # Events zuzuordnen
            # Das Event bedeutet hier, dass ein Klich auf das Feld
            # self.toggle(x, y) aufruft
            self.game.tag_bind(field, '<Button-1>', lambda event,
            x=x, y=y: self.toggle(x, y))

def toggle(self, x, y):
    # Beim Klicken die angrenzenden Kacheln umschalten
    # For-Schleife iteriert über die angrenzenden Kacheln
    for dx, dy in [(0, 0), (-1, 0), (1, 0), (0, -1), (0, 1)]:
        # ny und ny sind die Koordinaten der jeweiligen Nachbarkachel
        nx, ny = x + dx, y + dy
        # Kachel muss innerhalb des Rasters liegen
        # Es wird nur umgeschaltet, wenn die Kachel innerhalb des
        # Spielbretts liegt
        if 0 <= nx < GRID_SIZE and 0 <= ny < GRID_SIZE:
            self.states[ny][nx] ^= 1
```

```
                self.game.itemconfig(self.fields[(nx, ny)],
                fill=self.get_color(nx, ny))

        # Zähler erhöhen (+= 1) und Label aktualisieren
        self.move_count += 1
        self.clicks_label.config(text=f"Züge: {self.move_count}")

        # Nach jedem Zug prüfen, ob das Spiel gewonnen wurde
        if self.check_win():
            self.show_win_message()

    def get_color(self, x, y):
        # Gibt die Farbe der Kachel x y basierend auf dem Zustand zurück
        return "yellow" if self.states[y][x] == 1 else "black"

    def check_win(self):
        # Prüfen, ob das Spiel gewonnen wurde
        # (alle Kacheln sind 0 (schwarz))
        # all ist eine Funktion, die True zurückgibt, wenn alle Elemente
        # in der Liste 0 sind
        return all(not cell for row in self.states for cell in row)

    def show_win_message(self):
        # Erstellt ein neues Fenster, mit dem Titel "Gewonnen!"
        win_popup = tk.Toplevel(self.root)
        win_popup.title("Gewonnen!")
        # Kurze Nachricht, in wie vielen Zügen Spiel gewonnen wurde
        tk.Label(win_popup,
                text=f"Du hast in {self.move_count} Zügen gewonnen!",
                font=("Arial", 14)).pack(pady=10)
        # Lambda um Popup zu schließen und gleichzeitig das Spiel
        # zurückzusetzen
        tk.Button(win_popup, text="OK", command=lambda:
        [self.reset(), win_popup.destroy()]).pack(pady=5)

    def solve(self):
        print("Lösen")
        # Solver mit Zustandsmatrix aufrufen
        solutionMatrix = self.solver.solve(self.states)[0]
        # KeyMatrix ist die Lösung, welche Felder umgeschaltet
```

```
        # werden müssen
        # For-Schleife iteriert über die Felder der Lösung
        for y in range(GRID_SIZE):
            for x in range(GRID_SIZE):
                # Wenn die Kachel in der KeyMatrix 1 ist
                # (also geschaltet werden muss)
                if solutionMatrix[y][x] == 1:
                    # Dann rufe die toggle Funktion auf
                    # diesem Feld auf
                    self.toggle(x, y)
                    # Kurze Pause, damit es ersichtlich ist,
                    # welches Feld geschaltet wurde
                    time.sleep(1/GRID_SIZE)
                    # Nach jedem Schalten muss die GUI
                    # (self.root) aktualisiert werden
                    self.root.update()

    def reset(self):
        print("Zurücksetzen")
        # Züge zurücksetzen
        self.move_count = 0
        # Label aktualisieren
        self.clicks_label.config(text=f"Züge: {self.move_count}")

        self.round_count += 1
        self.rounds_label.config(text=f"Runde: {self.round_count}")

        # Neu generierte Zustände für die jeweiligen Felder
        self.states = self.generate_states()
        self.build_grid()


if __name__ == "__main__":
    root = tk.Tk()
    app = LightsOut(root)
    root.mainloop()
```

Programm 2 `Solver.py`

```
import numpy as np
import time

class Solver:
    def __init__(self, size):
        self.size = size
        self.n = size * size
        # Übergangsmatrix initialisieren
        self.transitionMatrix = self.build_matrix()

    def build_matrix(self):
        # Übergangsmatrix (Welche Lampe schaltet welche Lampen)
        # erstellen
        A = np.zeros((self.n, self.n), dtype=int)
        for i in range(self.n):
            A[i][i] = 1
            row, col = divmod(i, self.size)
            for r, c in [(row-1, col), (row+1, col),
                         (row, col-1), (row, col+1)]:
                if 0 <= r < self.size and 0 <= c < self.size:
                    A[i][r * self.size + c] = 1
        return A

    # Inverse der Übergangsmatrix in mod2
    def get_solutionMatrix(self, A, b):
        n = len(b)
        for i in range(n):
            if A[i][i] == 0:
                for j in range(i+1, n):
                    if A[j][i] == 1:
                        A[[i, j]] = A[[j, i]]
                        b[i], b[j] = b[j], b[i]
                        break
            for j in range(i+1, n):
                if A[j][i] == 1:
                    A[j] ^= A[i]
                    b[j] ^= b[i]
        # Rückwärtseinsetzen
        x = np.zeros(n, dtype=int)
```

```
        for i in reversed(range(n)):
            x[i] = (b[i] ^ sum(A[i][j] & x[j] for j
                               in range(i+1, n))) % 2
        return x


    # Prüft ob die Zustandsmatrix lösbar ist
    # Gibt die Lösung uns die Lösbarkeit zurück
    def solve(self, grid):
        # Startzeit für die Berechnung
        timeStart = time.time()
        isSolvable = False
        # Zustandsmatrix, die gelöst werden soll
        # 2D Array in 1D numpy Array umwandeln
        stateVector =
        np.array([cell for row in grid for cell in row], dtype=int)
        # Lösungsmatrix für gegebenes Feld
        solutionMatrix = self.get_solutionMatrix(self.transitionMatrix.copy(),
                             stateVector.copy())

        # Überprüfen ob die Lösung gültig ist
        # Übergangsmatrix * Lösung + Startzustand = Nullvektor b
        # M x + a = b (Test auf Lösbarkeit)
        testMatrix = np.add(np.linalg.matmul(self.transitionMatrix,
                                 solutionMatrix), stateVector)
        np.mod(testMatrix, 2, out=testMatrix)

        # Ist true, wenn die errechnete Lösung nur aus 0en besteht
        # Wenn true, dann ist das Feld lösbar
        allZeros = not np.any(testMatrix)

        # Ist true, wenn die Startkonfiguration nicht nur 0en enthält
        # Dann wäre das Spielbrett bereits gelöst
        startMatrix = np.any(stateVector)

        # Wenn beide Bedingungen erfüllt sind, dann ist das Spielbrett
        # lösbar
        if allZeros and startMatrix:
            isSolvable = True
```

```
        # 1D Array in 2D umwandeln
        solutionMatrix = np.reshape(solutionMatrix, (self.size, self.size))

        timeEnd = time.time()
        return[solutionMatrix, isSolvable, timeEnd - timeStart]
```