Lichter aus

Einleitung

Geschichte

Lichter aus ist ein Logik-Puzzlespiel welches in der Form das erste Mal in 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 ein Kachel jeweils sich selbst, und alle direkt an der Kachel liegenden Nachbarn um.

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 in der Zeile darüber diese aus.
Wenn nur Kacheln in der letzten Zeile leuchten gibt es verschiedene Kombinationen von Kacheln die man in der ersten Zeile wieder einschalten muss. Das wird dann wiederholt, bis alle Felder ausgeschaltet sind

Aufgabenstellung

Ziel der Aufgabe war es das Spiel **Lichter Aus** für ein 5x5 Lampen großes Spielfeld zu programmieren.
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 technischer Vorgaben zwischen einer `einfach` und `komplex` Version getrennt.
Diese unterscheiden sich 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 `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.

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 an sich benötigt werden.

Das 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.

```
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
            # color Variable wird basierend auf dem Zustand der Kachel aus dem states Arra
            color = 'yellow' if self.states[y][x] == 1 else 'black'
            # Für jede Koordinate wird eine Kachel mit Tkinter erstellt
            # Die color Variable wird Füllfarbe der jeweiligen Kachel
            field = self.game.create_rectangle(x * FIELD_SIZE, y * FIELD_SIZE, (x + 1) * FI
            # Anschließend wird die toggle Funktion für den Klick auf die Kachel zugewies

    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
        self.move_count += 1
            self.clicks_label.config(text=f"Züge: {self.move_count}")


    def
```

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 der 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 nimmt 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

Implementiert ist dieser Test ebenfalls in der `generate_states` Funktion.

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 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'.

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

Zustand	XOR	Ergebnis
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.

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 komplexte 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:

0	**1**
1	**0**

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.

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

Somit ergibt sich für unser 2x2 Spielbrett folgende Lösung:

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

Das heißt, um das als M definierte Spielbrett zu lösen muss man Feld 1 und Feld 4 umschalten.

Modulare Arithmetik

Die modulare Arithmetik kennen wir zum Beispiel von der Uhrzeit. 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
16 % 12 = 4 # <- Da 16/12 = 1 Rest 4
```

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 3 würde wieder eine 0 werden.

Implementierung

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

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)]:
                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.

TODO: Weiter erklären

Im nächsten Schritt wird die Inverse der Übergangsmatrix mithilfe des Gausschen Elimination ermittelt

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.

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)

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)

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 zur Lösbarkeit des 5x5 Feldes

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

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)

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)