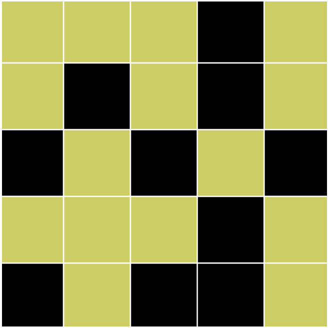
Turn all the lights off

ACM ICPC

Simon Hanrath

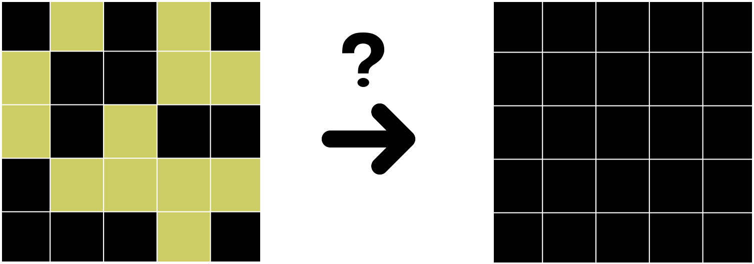
Story

We have to turn off all lights
in the office



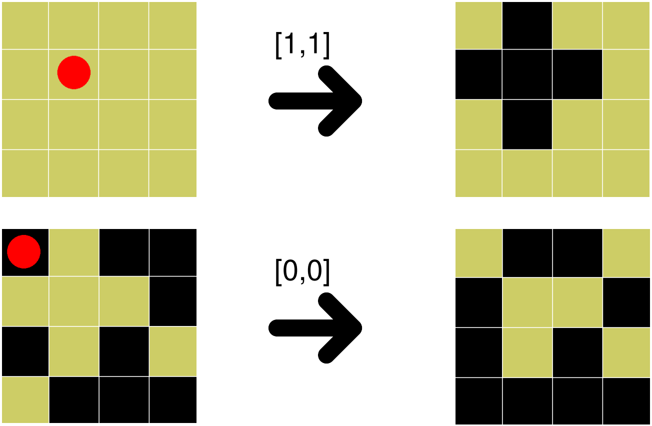
Bird's-eye view of the office

Our Task



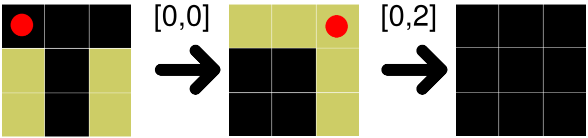
There Is a Catch

The lights are wired in a special way

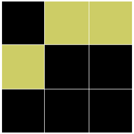


Quick Example

A quick example of a solvable configuration



Example of a non-solvable configuration



Specifications

Input

- $n \times n$, $n \in \{1, 2, .., 40\}$ Office
- Each room has exactly one light
- Each of the n^2 lights is either on or off

Output

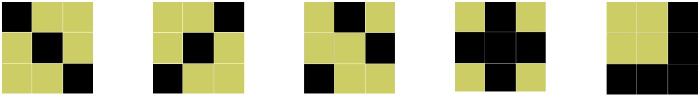
- Can all lights of this configuration be deactivated by performing only allowed actions

Getting to Know the Problem

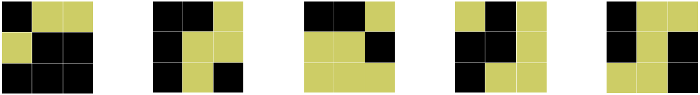
- I wrote a simulation in Pygame to play around with some configurations
- Starting with a non-solvable configuration generates new non-solvable configurations
- Starting with a solvable configuration generates new solvable configurations

Obvious Solution?

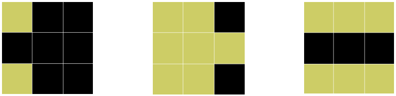
Solvable configurations



Non-solvable configurations



Unfortunately No Obvious Solution



...

Are also non-solvable

- This is harder than I expected!

Brute Force?

So just brute force the Problem, right?

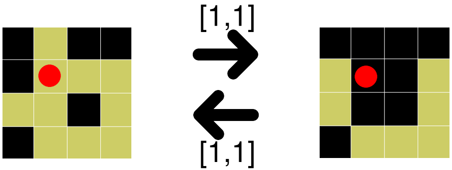
- The path to the solution can be of any length and is therefore basically impossible to brute force

Can the path really be of any length?

Can we somehow shorten the space of relevant paths?

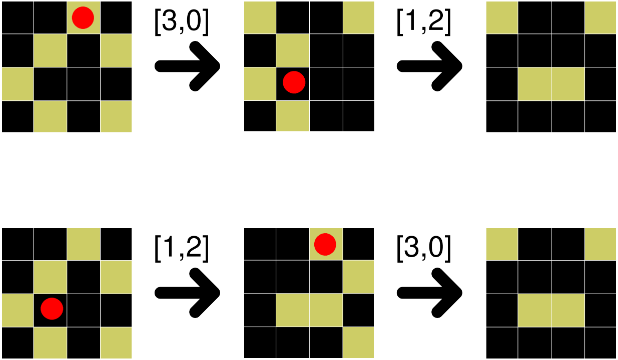
We Are in the Space Modulo 2

It is unnecessary to press a switch more than once!



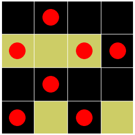
Order Does Not Matter

Every configuration depends only on the flipped switches, no matter in what order!



Can We Now Use Brute Force?

The solution (if one exists) can be reached by flipping m of the $n \times n$ switches!



can for example be represented as
$[0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0]$

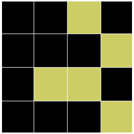
We "only" need to test $2^{(n^2)}$ combinations

For $n = 40$ that is still $4.446241648 \cdot 10^{482}$ combinations

Linear Algebra

Can we formulate the problem mathematically?

The office can easily be represented as a $n \times n$ matrix



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

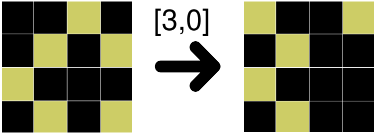
Linear Algebra

Flipping the switch at position i,j can also be represented as a $n \times n$ matrix!

$$A_{1,1} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad A_{2,0} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Linear Algebra

Now we can mathematically formulate the possible actions



corresponds to

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

Linear Algebra

We want to go from a given configuration C to the zero matrix.

$$\begin{bmatrix}1&1&0&1\\1&0&1&0\\0&1&1&1\\1&0&1&0\end{bmatrix} + b_{0,0} \cdot \begin{bmatrix}1&1&0&1\\1&0&0&0\\0&0&0&0\\1&0&0&0\end{bmatrix} + \ldots + b_{3,3} \cdot \begin{bmatrix}0&0&0&1\\0&0&0&0\\0&0&0&1\\1&0&1&1\end{bmatrix} = \begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{bmatrix}$$

$$\Leftrightarrow$$

$$C + \sum_{i,j}(b_{i,j} \cdot A_{i,j}) = 0$$

$$\Leftrightarrow$$

$$\sum_{i,j}(b_{i,j} \cdot A_{i,j}) = C$$

Linear Algebra

We can represent the action matrices as well as the office configuration matrix as vectors.

$$A_{1,1} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \rightarrow A_{1,1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}^T$$

We still want to solve the same equation

$$\sum_{i,j}(b_{i,j} \cdot A_{i,j}) = C$$

Linear Algebra

Now we can rewrite the equation

$$\sum_{i,j}(b_{i,j} \cdot A_{i,j}) = C \quad \Leftrightarrow \quad A \cdot b = C$$

with

$$A = \begin{bmatrix} A_{0,0} & \mid & A_{1,0} & \mid & \dots & \mid & A_{n-1,n-1} \end{bmatrix}$$

$$b = \begin{bmatrix} b_{0,0} & b_{1,0} & \dots & b_{n-1,n-1} \end{bmatrix}$$

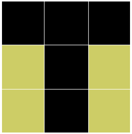
$$C = \begin{bmatrix} C_{0,0} & C_{1,0} & \dots & C_{n-1,n-1} \end{bmatrix}$$

Linear Algebra

To determine whether a solution for a configuration C exist we just have to figure out if a b exists such that $A \cdot b = C$!

➜ This can for example be done with the Gauß Jordan Method.

Example of the Final Algorithm



Is the this configuration solvable?

1. Create A and C

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$A_{0,0}$ $A_{2,2}$

Example of the Final Algorithm

2. Use the Gauß Jordan algorithm in mod 2

- You start with $k = 0$
- If $A_{k,k} = 0$, swap row k with one below that has a 1 at position k
- If $A_{k,k} = 1$, then for each row that has a 1 at position k and is not row k itself, add the row k
- Now there is only one 1 in column k (at position k)
- Is $k < n^2 - 1$, $k+ = 1$ and repeat

Each modification (addition or swapping) of the rows of A is also done with the vector C

Example of the Final Algorithm

After applying the Gauß Jordan algorithm to our example:

$$A' = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad C' = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Example of the Final Algorithm

3. Evaluate the result

- All $A'_{k,k} = 1$ -> exactly one solution
- $\exists k : A'_{k,k} = 0$ and $C'_k = 1$ -> no solution
- $\neg\exists k : A'_{k,k} = 0$ and $C'_k = 1$ -> multiple solutions

So we just have to check whether $\exists k : A'_{k,k} = 0$ and $C'_k = 1$

Implementation in Python

Why Python?

- Python is not Java
- Python has numpy
- Python is easy to read and write
- Python has Pygame

The Code is pretty straight forward, but we can still talk about it after the presentation

Key Findings

- We are in the modulo 2 space
- The order of actions does not matter

➜ therefore a solution must be reachable by flipping m out of the $n \times n$ switches.

- Now we can reduce the problem to deciding if an LGS is solvable

➜ This can be done with Gauß Jordan elimination

Code Slides

```python
def is_solvable(input_file):
    """
    Reads from file and decides whether given grid is solvable by allowed actions
    (by checking whether an Ax=b equation is solvable)

    Args:
        input_file: .txt file that specifies the start grid

    Returns:
        "Yes" or "No" depending on the configuration (solvable non-solvable)
        """
    b = generate_grid_matrix(input_file).flatten()
    A = generate_A(int(math.sqrt(len(b))))
    n = len(A)

    # main loop
    for k in range(n):
        # partial pivoting
        if A[k, k] == 0:
            #check if we have a one belowe the 0 at k,k
            for i in range(k + 1, n):
                if A[i, k] == 1:
                    #if we have a 1 switch the two rows
                    A[[k, i], :] = A[[i, k], :]
                    b[k], b[i] = b[i], b[k]
                    break

        # elimination
        for i in range(n):
            if i == k or A[i, k] == 0: continue
            # if we have a 1 at n,k add row k to n, ow we have a 0 at n,k
            A[i, k:n] = (A[i, k:n] + A[k, k:n]) % 2
            b[i] = (b[i] + b[k]) % 2

    # check if solution exists
    for i in range(n):
        if A[i, i] == 0 and b[i] == 1:
            return "No"
    return "Yes"
```

Code Slides

```
def simulate_click(grid, pos):
    """
    Simulates the flipping of a switch at given position and grid

    Args:
        grid: 2D np.array that represents the grid
        pos: python list with x and y coordinate of the flipped switch

    Returns:
        2D np.array that represents the grid after the click at given position
    """
    x = pos[1]
    y = pos[0]
    grid_size = len(grid)

    if x < grid_size and y < grid_size:
        grid[x][y] = not grid[x][y]
        grid[(x + 1) % grid_size][y] = not grid[(x + 1) % grid_size][y]
        grid[(x - 1) % grid_size][y] = not grid[(x - 1) % grid_size][y]
        grid[x][(y + 1) % grid_size] = not grid[x][(y + 1) % grid_size]
        grid[x][(y - 1) % grid_size] = not grid[x][(y - 1) % grid_size]

    return grid
```

Code Slides

```
def generate_A(grid_size):
    """
    Generates the Action Matrix for a given Grid size

    Args:
        grid_size: int that represents the size of the grid

    Returns:
        2D np.array that represents the Action Matrix
        (Matrix of all possible actions for a Grid)
    """
    A = np.zeros([grid_size ** 2, grid_size ** 2], int)
    counter = 0
    for y in range(grid_size):
        for x in range(grid_size):
            action = np.zeros([grid_size, grid_size], int)
            A[counter, :] = simulate_click(action, [x, y]).flatten()
            counter += 1
    return A
```

Code Slides

```python
def generate_grid_matrix(challenge_input_file):
    """
    Reads from file to create the desired grid

    Args:
        challenge_input_file: .txt file that specifies the start grid
    Returns:
        2D np.array that represents the Grid Matrix
    """
    file = open(challenge_input_file, "r")
    size = int(file.readline())
    matrix = np.zeros([size, size], int)
    file.readline()
    for line in file:
        pos = [int(x) for x in line.split()]
        matrix[pos[0]][pos[1]] = 1
    file.close()
    return matrix
```