# Generating and solving the LIGHTS OUT! game in first order logic

Borbála Fazakas, Beáta Keresztes, Adrian Groza

Technical University of Cluj-Napoca
Department of Computer Science

September, 2022

# The Lights Out! Game - Rules

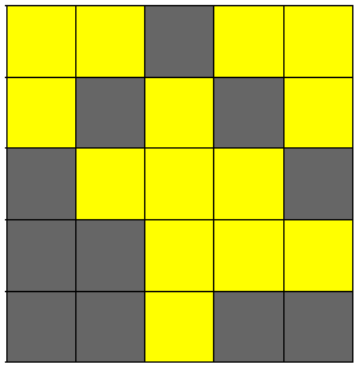
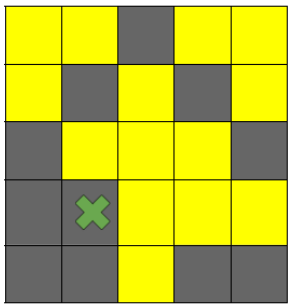


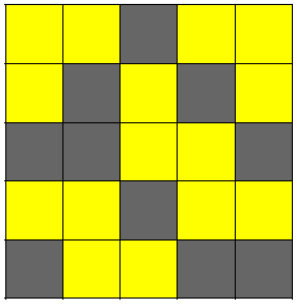
Figure: Sample Game Configuration $S_0$

- State: grid of tiles, some of them lit up, the others are turned off
- Goal: turn all lights off
- Actions: clicking on one tile, which toggles the state of the clicked tile and its neighbors

# Sample Action, considering a cross neighboring pattern



(a) $S_0$



(b) State after clicking (3, 1)

Our goal: desktop application for the LightsOut! game, which can

- generate solvable LightsOut puzzles
- solve the puzzles, in order to give hints to the user

Tools for FOL:

- Prover9: theorem prover
- Mace4: finite model finder

Two approaches to model the game:

- as a planning-task
- as a model-finding task

# Solving the LightsOut! Game as a Planning Task

Given the **initial state**, find a sequence of **actions** that leads to the **goal state**.

- **state representation**: for an n $\times$ n grid, use a 1 $\times$ $n^2$ boolean list V, with $V[i] = 0$ meaning the light bulb in row $i/n$ and column $i\%n$ is off, and $V[i] = 1$ for light on.
  - for example, for the previously presented game configuration $S_0$

$$V = s([1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0])$$

  - in particular, the goal state for a 5 $\times$ 5 grid is represented by

$$V = s([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])$$

# Solving the LightsOut! Game as a Planning Task

## Actions - Neighboring Patterns

- **action representation**: neighboring pattern + the effect of toggling
  - neighboring pattern: n(X, Y) is true $\Leftrightarrow$ if X gets clicked, then Y's state is toggled
    - sample neighboring pattern: the cross-like pattern

$$nCross(x, y) \leftrightarrow$$
$$(x \neq n \wedge x \neq 2n \wedge ..... \wedge x \neq n^2 \wedge x + 1 == y) \vee$$
$$(y \neq n \wedge y \neq 2n \wedge ... \wedge y \neq n^2 \wedge y + 1 == x) \vee$$
$$x == y \vee x + n == y \vee y + n == x$$

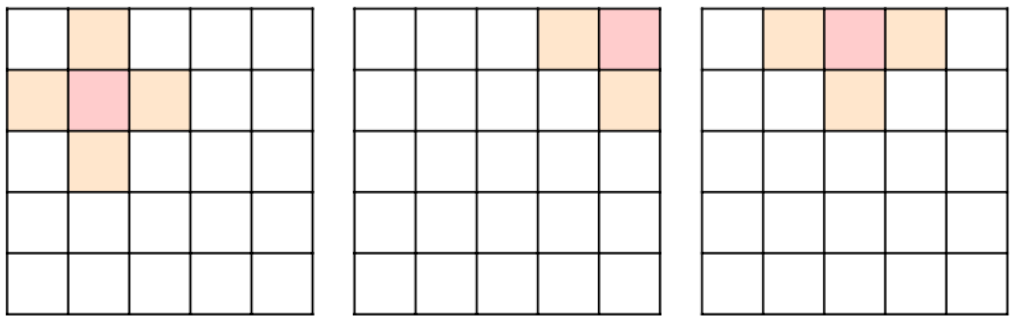


Figure: nCross

# Solving the LightsOut! Game as a Planning Task

## Actions - Toggling Effect

- **action representation**: neighboring pattern + the effect of toggling
  - toggling effect: a function that describes what the next state is if tile S gets clicked in the state [S:R]

$$\begin{aligned} toggleVector([], S, C) &= []. \\ toggleVector([F:R], S, C) &= if(n(S, C), \\ &\quad [toggle(F) : toggleVector(R, S, C+1)], \\ &\quad [F : toggleVector(R, S, C+1)]). \end{aligned} \tag{1}$$

  - $[F : R]$ = the current state
  - S = the clicked tile's number
  - C = the index of F in the original state array

The production rule describing the action of clicking tile k:

$$state([F:R]) \rightarrow state(toggleVector([F:R], k, 1)) \tag{2}$$

($n^2$ such production rules needed)

# Solving the LightsOut! Game as a Planning Task

- The neighboring predicate, toggleVector functions and the production rules fully describe the game and allow Prover9 to find the solutions, **theoretically**
- In practice, this game description leads to a breadth-first search solution, with, for an n $\times$ n grid, $n^2$ options to try at each step
- For an initial state with the shortest solution longer than 6 steps, Prover9 fails to find a solution with a memory limit of 1000MB $\rightarrow$
- Redundant steps: it is redundant to switch a light bulb twice $\rightarrow$ if a solution for a puzzle exists, then there must be a solution in which any light bulb should be switched maximum once
- The final state of a light bulb is given by the parity of the total number of switches applied on it or on its adjacent light bulbs. If the light bulb is toggled an even number of times, then its final state is the same as its original state. If it is toggled an odd number of times, then its final state is different from its original state.
- The order of the steps is irrelevant.

# Solving the LightsOut! Game as a Model Finding task

## Starting point

- Instead of looking for a sequence of steps that leads to the goal state, we can search for a set of steps.
- The task reduces to finding a sequence X of nodes (activation set), such that activating all nodes in X will turn off all nodes in G.
- Given to Mace4: the initial State, on(X, Y) $= 1 \leftrightarrow$ the light bulb on row X and column Y is on originally
- Found by Mace4: the set of steps, switch(X, Y) $= 1 \leftrightarrow$ clicking on the light bulb in row X and column Y is part of the solution
- The key: any light bulb (X, Y) for which on(X, Y) $= 0$ must be toggled an even number of times, and the others an odd number of times

$$oddToggles(X, Y) \iff$$
$$Xor(Xor(Xor(Switch(X+1, Y), Switch(X-1, Y)),$$
$$Xor(Switch(X, Y-1), Switch(X, Y+1))), Switch(X, Y))$$

$$oddToggles(X, Y) \iff On(X, Y)$$

# Solving the LightsOut! Game

## Comparison

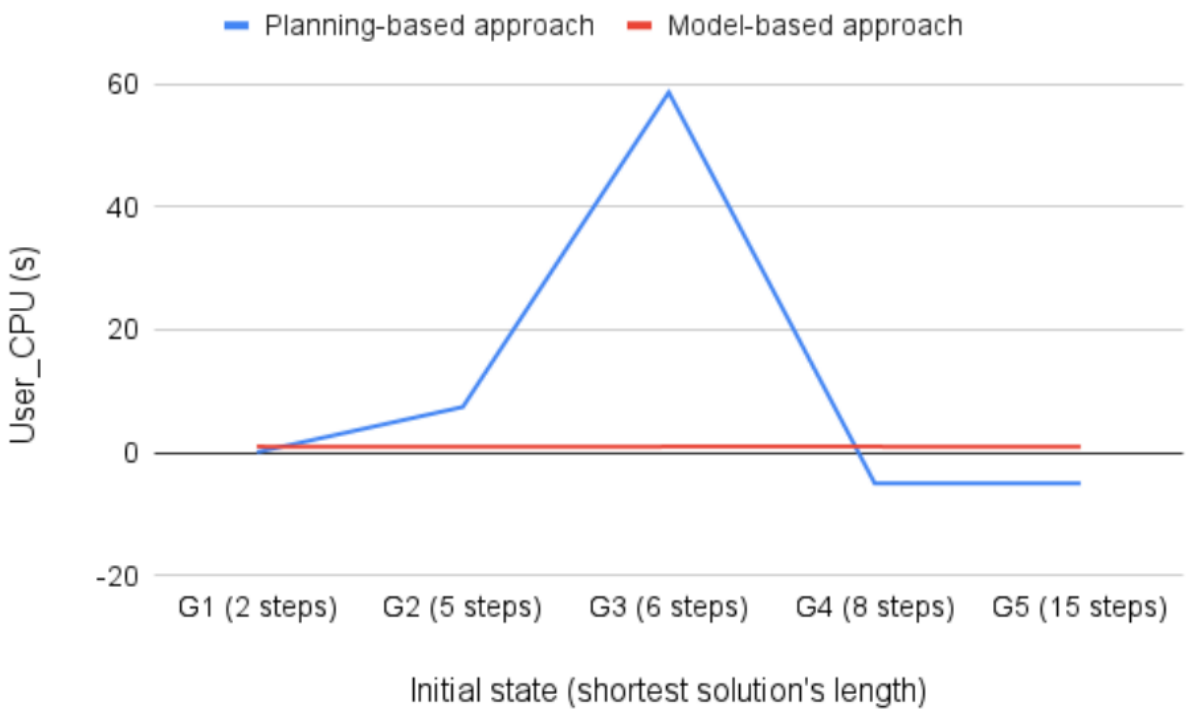



Figure: The comparison of the planning-based and the model finder-based solutions

Note that the negative User_CPU(s) values show that the execution was unsuccessful with a memory limit of 1000MB.

# Generating LightsOut! games using FOL

Goal: generate **solvable** game configurations.
Three approaches:

1. Using a puzzle-solving algorithm
2. Using linear algebra and a model generator
3. Using an improved linear algebra-based method

Key Idea: use the previous puzzle-solver to solve the reverse task: let the puzzle-solver find game configurations that it can solve.
Modifications in the Mace4 implementation:

- do not give values to the *on(x, y)* matrix
- specify that *on(x, y)* is a boolean matrix
- to avoid the trivial game, specify that *on(x, y)* has at least one true value:

$$\exists x \; \exists y \; on(x, y) = 1 \tag{3}$$

- this solution does more than what we need
- the same puzzle gets generated multiple times (once for each of its solutions)

# A linear algebra-based model generator

The second solution is based on the work of Anderson and Feil.

- Initial game state = the column vector $B$, where $B[i] = 0$ iff the light bulb in row $i/n$ and column $i\%n$ is off, and $B[i] = 1$ for light on, with

$$\vec{B} = On(0), On(1), On(2), \ldots On(n \times n) \tag{4}$$

- Recall that the function $On(n)$ states that the bulb $n$ is turned on, with $On(x) = 1 \vee On(x) = 0$.

## Theorem

*$B$ is a solvable game state* $\iff$ $B \perp N_1$ & $B \perp N_2$ *where*

$$\vec{N_1} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & & & & & \end{pmatrix} \tag{5}$$

$$\vec{N_2} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & & & & & \end{pmatrix} \tag{6}$$

# A linear algebra-based model generator

**Theorem**

*Each solvable game state has exactly 4 solutions.*

**Theorem**

*If* $\vec{X} = \begin{pmatrix} Switch(0) & Switch(1) & Switch(2) & \dots & Switch(24) \end{pmatrix}$ *is a solution for game state B, then the 4 solutions are*

$$\begin{aligned} \vec{X_1} &= \vec{X} \\ \vec{X_2} &= \vec{X} + \vec{N_1} \\ \vec{X_3} &= \vec{X} + \vec{N_2} \\ \vec{X_4} &= \vec{X} + \vec{N_1} + \vec{N_2} \end{aligned} \tag{7}$$

We ask Mace4 to find the boolean vector $On$ (= the initial game state), such that $On \perp N_1$ & $On \perp N_2$.

# A linear algebra-based model generator

## Applying the theorems

- verify the perpendicularity by computing the dot products:

$$(x > 0 \wedge y + 1 = x \wedge Z = DotProd_1(y) \wedge O = On(x) \wedge N = N_1(x)) \rightarrow DotProd_1(x) = Xor(Z, And(O, N)). \quad (8)$$

$$DotProd_1(0) = 0. \quad (9)$$

$$(x > 0 \wedge y + 1 = x \wedge Z = DotProd_2(y) \wedge O = On(x) \wedge N = N_2(x)) \rightarrow DotProd_2(x) = Xor(Z, And(O, N)). \quad (10)$$

$$DotProd_2(0) = 0. \quad (11)$$

So the conditions for perpendicularity can be expressed as $DotProd_1(n^2 + 1) = 0$ and $DotProd_2(n^2 + 1) = 0$ (array indexed from 1)

We ask Mace4 to find the boolean vector $On$ (= the initial game state), such that $On \perp N_1$ & $On \perp N_2$.

- $On(0) = 0$ (recall that the arrays are indexed from 1)
- $\exists x\ On(x) = 1$, to avoid the trivial game

# A a linear algebra-based model generator

Advantages:

- we search only for a model for the initial state, not for the solution

Disadvantages:

- for an $n \times n$ game, we need to use the arithmetic library with a domain size of $n^2$, to enable using the $n \times n$-long vector $On$. The execution fails even for a $5 \times 5$ game, with a memory limit of 1000MB

Key Idea: by simply reorganizing the $n^2$-long vectors into $n \times n$ matrices, we can reduce the domain size of $n^2$ to n.

# Improving linear algebra-based model generator

The relations for computing the dot products become (for a $5 \times 5$ game):

$$(y \geq 1 \wedge yn + 1 = y \wedge d = DotProd_1(x, yn) \wedge$$
$$o = On(x, y) \wedge n = N_1(x, y)) \rightarrow$$
$$DotProd_1(x, y) = Xor(d, And(o, n)).$$
$$(x \geq 1 \wedge xn + 1 = x \wedge d = DotProd_1(xn, 4) \wedge$$
$$o = On(x, 0) \wedge n = N_1(x, 0)) \rightarrow$$
$$DotProd_1(x, 0) = Xor(d, And(o, n)).$$
$$0 = On(0, 0) \wedge n = N_1(0, 0) \rightarrow$$
$$DotProd_1(0, 0) = And(O, N).$$
$$(y \geq 1 \wedge yn + 1 = y \wedge d = DotProd_2(x, yn) \wedge$$
$$o = On(x, y) \wedge n = N_2(x, y)) \rightarrow$$
$$DotProd_2(x, y) = Xor(d, And(o, n)).$$
$$(x \geq 1 \wedge xn + 1 = x \wedge d = DotProd_2(xn, 4) \wedge$$
$$O = On(x, 0) \wedge n = N_2(X, 0)) \rightarrow$$
$$DotProd_2(x, 0) = Xor(d, And(o, n)).$$

# Improving linear algebra-based model generator

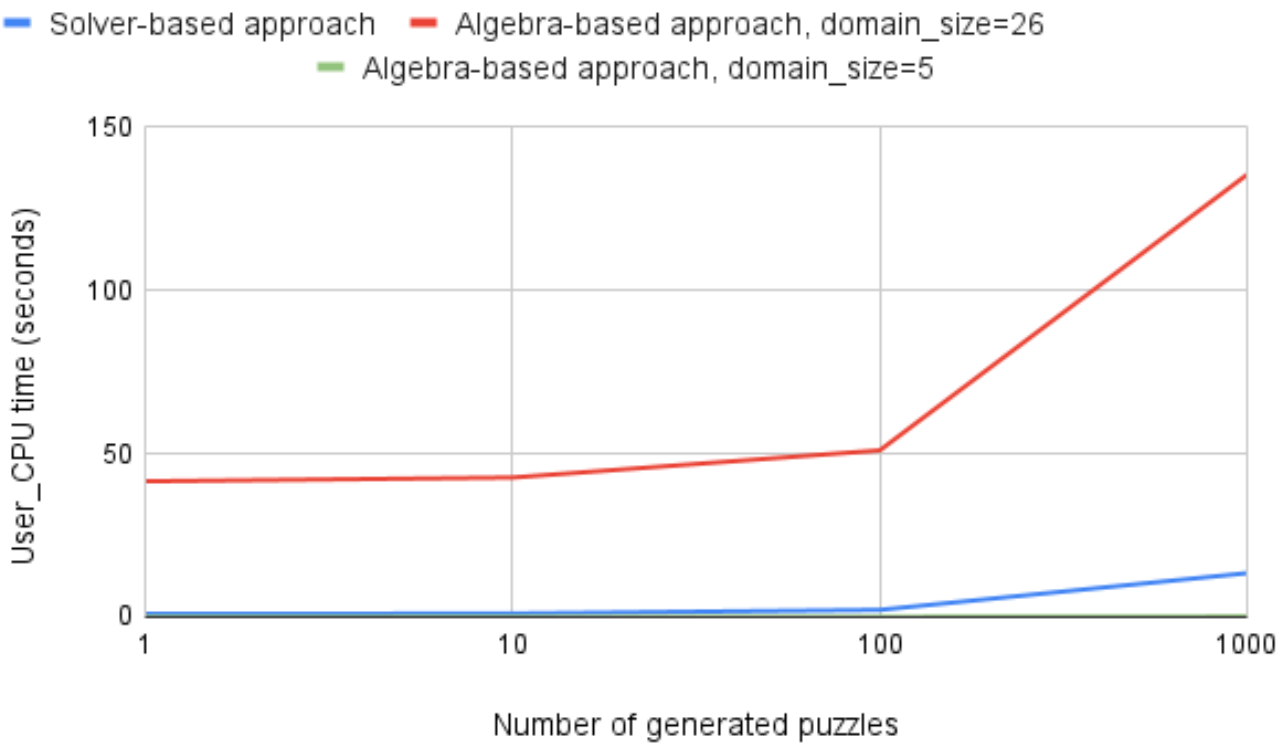
## Implementation

Mace4 has to find the values of $On(x, y)$ with three constraints:

$$On(x, y) = 1 \vee On(x, y) = 0 \tag{12}$$

$$\exists x \; \exists y \; On(x, y) = 1 \tag{13}$$

$$DotProd_1(4, 4) = 0 \wedge DotProd_2(4, 4) = 0 \tag{14}$$



# Generating LighsOut! games

## Comparison

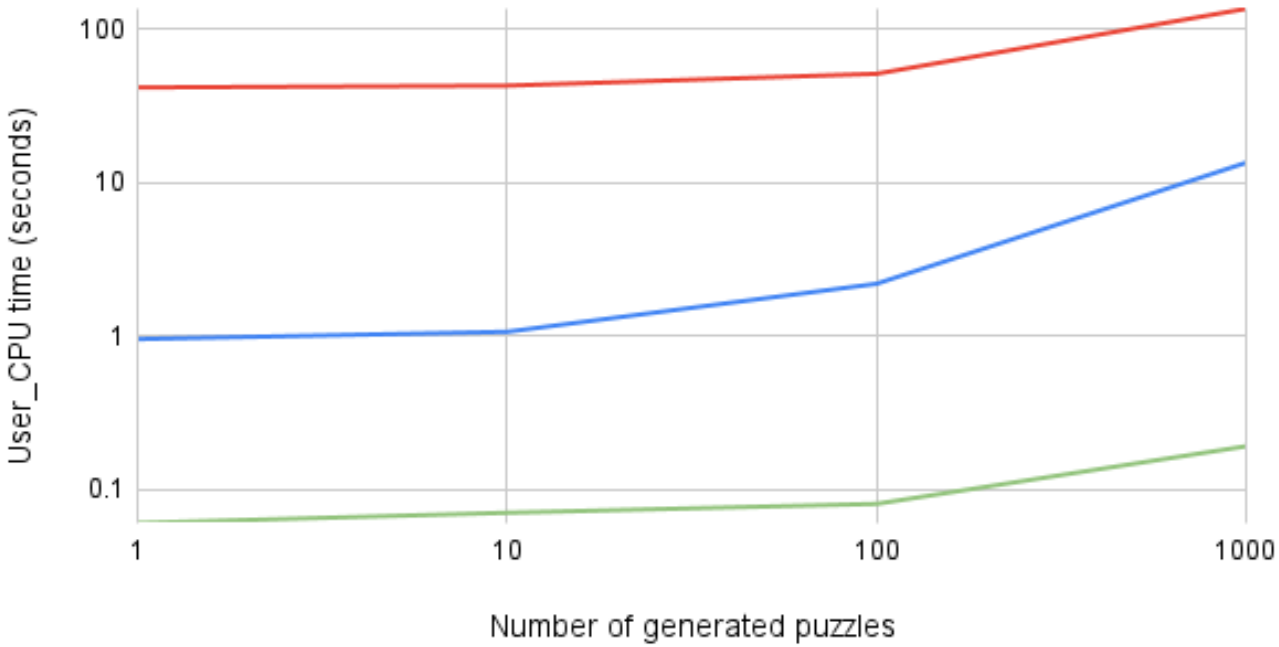



Figure: Performance Comparison

Note the logarithmic scale

# Conclusions

- LIGHTS OUT! games can be solved both by planning in FOL, with Prover9's production mode, and through model finding methods.
- While Prover9's production mode is a convenient way for solving puzzles, it doesn't allow us to exploit any redundancies and symmetries. Thus, Mace4 is more efficient as a game solver.
- Solvable LIGHTS OUT! games can be generated based on game solvers and based on model finders relying on linear algebra as well.
- Game generators based on linear algebra are more efficient, because they are based on an equivalent criterion for the existence of a solution, but do not intend to find that solution.
- Fifth, reducing the domain size in Mace4 can have a significant impact on the performance of the program.