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

Borbála Fazakas[1], Beáta Keresztes[1] and Adrian Groza[1]

***Abstract*—We compare here declarative approaches to model the LIGHTS OUT! game and its friends in First Order Logic (FOL). First, we solve the game using: (i) planning in FOL and (ii) a model finder for finite domains, for which we rely on Prover9 and Mace4. Second, we show how LIGHTS OUT! puzzles can be automatically generated by reasoning on finite models of FOL theories. We designed three solutions: (i) using a LIGHTS OUT! game solver in FOL, (ii) using linear algebra and a model generator, and (iii) improving the linear algebra-based method by decreasing the domain size. Third, we show how declarative knowledge can be reused to solve and generate different extensions of the LIGHTS OUT! game. We experimentally compare the proposed declarative methods and we discuss some extensions of theLIGHTS OUT! game.**

***Index Terms*—Modelling in First Order Logic; Satisfiable models; Planning in FOL; LIGHTS OUT! game;**

I. INTRODUCTION

Logical games are an efficient instrument to engage students in learning artificial intelligence from various perspectives including pedagogical [1], [2], mind [3] or anthropological [4]. Modelling puzzles in First Order Logic (FOL) has been given some attention in the literature, with two examples being the 138 puzzles from the Thousands of Problems for Theorem Provers [5], and more recently the 144 puzzles from [6].

LIGHTS OUT! is a puzzle in which you are given a grid of tiles (or lights), some of them lit up, others turned off. The goal is to turn off all the lights in the grid by clicking on the tiles, in as few moves as possible. Each click toggles that light and its neighbours in a cross-like pattern. The player can click on any tile, and as an effect, it will toggle not only the state of the selected light but also that of its direct (non-diagonal) neighbours. The 5×5 electronic version was released by Tiger Electronics in 1995, following the 3×3 version called Merlin in 1970. Formally, the game is an undirected graph $G = (V, E)$ with n nodes where each node $v \in V$ has a state $B_v \in 0, 1$. We say v is "off" if $B_v = 0$ and "on" if $B_v = 1$.

When developing an interactive application, based on the LIGHTS OUT! game, two tasks are (1) to find solutions for the puzzles, in order to provide useful hints (tips) for the next move of the user, and (2) to generate solvable puzzles, for selecting a new game (board) layout. Thus, the paper not only addresses the solution finding of the game but also makes use of solution generation models to further create game instances. The quantitative evaluation is given to show the effectiveness of the models.

[1]Computer Science Department, Technical University of Cluj-Napoca, 28 Memorandumului, Cluj-Napoca, Romania keresztesbeata00@yahoo.com, fazakasbori@gmail.com, adrian.groza@cs.utcluj.ro

II. MODELLING THE LIGHTS OUT! GAME IN FOL

We present ways to solve the LIGHTS OUT! game, as well as generate solvable instances of the puzzle. Two approaches are described next in terms of solving a given puzzle, describing it as a *planning task*, or a *model finding task* in FOL. The planning-based approach utilizes production rules to construct sequences of steps that lead to the winning state. The model-finding approach, on the other hand, relies on three important observations regarding the game, which are subsequently translated into a task that attempts to find sets of light bulb selections to switch. A direct comparison between the two shows how model-finding scales better with the parameter values required to solve a given instance. Both approaches were formalised in Prover9 theorem prover and its Mace4 model finder [7].

A. Planning in First Order Logic

The game can be approached as a planning task in first order logic. Given a game position represented by the state of each light bulb in the 5x5 grid, to goal is to find a sequence of steps which lead to the winning state. The solution is a sequence of (x_i, y_i) steps, meaning that the i^{th} step of the solution is switching the light bulb in position (x_i, y_i) of the grid, and by applying all steps from 0 to $n-1$, where n is the solution length, we get to the winning state. The shortest solution is of interest to assist users by providing hints for solving the puzzle.

For planning in FOL, we use state-based reasoning to find a sequence of steps that lead to the winning state. As a planning task, we need to formalise in first order logic (i) the initial state, (2) the goal state, and (3) allowed steps and their effects.

First, for the initial state, the 5×5 boolean matrix is represented with a 1×25 list V, with $V[i] = 0$ meaning the light bulb in row $i/5$ and column $i\%5$ is off, and $V[i] = 1$ for light on. For example, the initial state of the game G_0 is:

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

Second, the goal state with all lights turned off represents the theorem to prove, formalised with:

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

Third, for modelling the possible moves, we need to formalise the neighboring pattern. In the classical version of the game, this pattern represents a cross (Fig. 1). The $nCross(x, y)$ predicate is true only if the light bulb x is the neighbor of

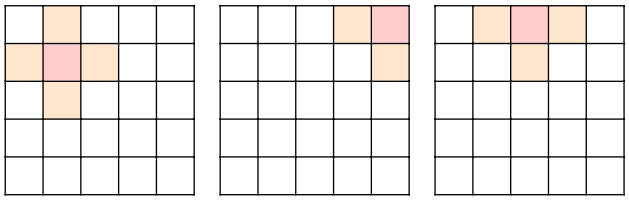


Fig. 1: The three cases for the cross neighboring pattern

light bulb y. For a $n \times n$ grid, the cross neighboring pattern is formalised for the corresponding list of length $n \times n$ with:

$$\begin{aligned} &nCross(x,y) \leftrightarrow \\ &(x \neq n \wedge x \neq 2n \wedge ..... \wedge x \neq n^2 \wedge x+1 == y) \vee \\ &(y \neq n \wedge y \neq 2n \wedge ... \wedge y \neq n^2 \wedge y+1 == x) \vee \\ &x == y \vee x+n == y \vee y+n == x \end{aligned}$$

Next, we formalise the effect of toggling a light bulb with a given state with the function $toggle(0) = 1$ and $toggle(1) = 0$. Since, each action affects maximum 5 cells, we define the predicate $toggleVector([F{:}R], S, C)$ with three variables : (i) the list $[F{:}R]$ containing the current state of the light bulbs, (ii) the identifier of the step S to be taken, $1 \leq S \leq 25$, and (iii) C, the index of F in the original vector describing the current state of the game, $1 \leq C \leq 25$ (note that the function is implemented recursively, and at each recursive step, 1 element is cut from the beginning of the list. This is why, for example, after 10 recursive steps, the F corresponds to the 10^th element of the 25-element list describing the current state of the game).

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

The predicate $toggleVector$ switches the state of the neighbors of the light C. The predicate $n(S,C)$ is the current neighboring pattern, e.g. $n(S,C) \leftrightarrow nCross(S,C)$. When playing the game with a different neighboring pattern (e.g. diagonal), this generic definition for the predicate $toggleVector$ works only by stating the current pattern, e.g., $n(S,C) \leftrightarrow nDiagonal(S,C)$. Also note that the Prover9 provides the if construct on top of first order logic.

At each step there are $n \times n$ possible actions, one for each button. For 5×5 grid, we used 25 production rules to move from one state to another. For instance, the rule activating when pressing the bulb in (0,0) is:

$$state([F:R]) \rightarrow state(toggleVector([F:R], 1, 1)) \quad (2)$$

While this code seems simple enough at the first sight, if we take into account that all Prover9 can do based on it is to apply a backtracking directly, with 25 options at each step and no clues for detecting earlier that from a given state S the goal state cannot be reached in the desired number of steps, it is clear, that this solution is highly inefficient. As an experiment, for the G_1 game on 5×5 grid below, the shortest solution consists of 5 steps: (0,0), (0,4), (2,2), (4,3), (4,4).

$$G_1 = \begin{bmatrix} \mathbf{1} & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & \mathbf{0} & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & \mathbf{1} \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

Prover9 takes around 8 seconds to find a solution, and uses 56 MBs of memory. This is not so surprising, given that Prover9 needs to try at least all the $25^4 > 2^{16}$ potential solutions of length 4 before finding the shortest solution of length 5. Moreover, for solving game G_2 below, with the shortest solution of 6 steps ((0,0), (0,1), (0,3), (0,4), (4,1), (4,4)), Prover9 takes around 59.6 seconds and 315MBs. These experiments were run on a virtual machine having an Intel(R) Core(TM) i7-8750H CPU of 2.20GHz max frequency, with 8.0GB RAM, hosting a 64-bit operating system, Ubuntu 20.04.

$$G_2 = \begin{bmatrix} \mathbf{0} & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & \mathbf{0} & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 1 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ \mathbf{1} & 1 & 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & \mathbf{1} \end{bmatrix},$$

We observed that this solution cannot be applied for any 5×5 grid with 7 or more steps in the shortest solution: the amount of memory required will be too large. Since, there are plenty of configurations with the shortest path having more than 8 steps, it is desirable to find an improved game solver algorithm. Such an improvement is by finding finite models of the LIGHTS OUT! formalisation in FOL.

Instead of planning, a FOL theory can be used to solve the game by finding interpretation models.

B. Finding models for the LIGHTS OUT!

Developing a LIGHTS OUT! solver based on interpretation models is based on the following observations:

1) *Redundant steps*: it is redundant to switch a light bulb twice, because that would cause its state to be set back to the original one. Hence, if a solution for a puzzle exists, then there must be a solution in which any light bulb should be switched maximum once.
2) *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*: the state of a light bulb is toggled exactly when it or an adjacent light bulb is switched. Overall, 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.
3) *The order of the steps is irrelevant*: based on observation 2, it follows naturally that for the final state of the light bulb only the number of the switches applied on it and on its neighbors matters, but not the order in which these switches were applied.

Based on observations 1 and 3, instead of searching for a solution represented by a sequence of steps, we may search for just a set of steps. Thus, the task reduces to finding a sequence $\mathcal{X}$ of nodes, called *the activation set*, such that activating all nodes in $\mathcal{X}$ will turn off all nodes in G. Remark that, toggling a light an even number of times would be redundant as it will end up in the same state as initially.

Whereas searching for a sequence of length n may require evaluating up to $25 * 24 * ...(26 - n)$ options, for set of steps out of 25 possible steps there are 2^{25} options overall (not considering the size of the set). Based on this simplification, we can avoid using Prover9 in production mode to generate a sequence of steps, and we can instead use Mace4 to generate a solution model. To this aim, we introduce the predicate $switch(x, y)$ being true iff switching the light bulb on position (x, y) is a step of the solution. We use also the function $on(x, y) = 1$ iff the light bulb on position x and y is "on" and $on(x, y) = 0$ if it is off. Note that $on(x, y)$ is a function instead of predicate, aiming to apply some functions on it. The predicate $oddToggles(x, y)$ marks that the number of toggles applied to a light bulb (i.e. the number of switches applied to that light bulb and its neighbors) is odd. We can define the predicate $oddToggles$ by computing the Xor-Sum of $switch(x, y)$, $switch(x - 1, y)$, $switch(x + 1, y)$, $switch(x, y - 1)$, and $switch(x, y + 1)$, because the Xor-Sum of some binary numbers gives the modulo 2 value of their sum.

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

Now, based on Observation 2, we can say that we want a light bulb to be toggled an odd number of times if initially it's turned on, and an even number of times if initially it's turned off, so that, in the end, all light bulbs will be turned off. Thus, $oddToggles(X, Y) \iff On(X, Y)$. The above described relations are enough to find a solution to an initial game state, provided as an assumption to Mace4 using the $on(x, y)$ function.

One point that was missed in the above train of thought is that some light bulbs do not have an adjacent light bulb in all 4 directions, so the definition of $oddToggles$ cannot be applied directly. To overcome this issue, all the matrices were extended with one additional row and one column in all sides, so that all cells of the original matrices had neighbors in all directions. It was fixed though, that the "light bulbs" in the extended rows/columns cannot be turned on or switched, so they didn't actually have an effect on the other light bulbs.

C. Comparing planning with model based approach

For comparing the planning with the model based approach, consider the five LIGHTS OUT! games G_1-G_5 in Table I. Each corresponding solution appears below the game. For instance, the game G_1 is solved by pressing two buttons: (0,0) and (0,4). The game G_5 is solved by pressing 15 buttons, as listed in bottom-right of Table I.

By asking for all the solutions, Mace4 returns 4 models for game G_1 (Table II). Here, values of one represent which bulb to press. Recall, the order has no relevance. By summing each matrix, we obtain the minim solution.

Running experiments have shown (Fig. 2), that while for the model based approach, the execution time remains constant with respect to the number of steps in the solution, and it takes a little less than 1s, the planning based approach execution time grows exponentially with the number of steps, takes almost 1 minute for a 6-step solution and fails to provide a solution with the memory limit of 1000MB for a puzzle with more than 6 steps in its shortest solution. Negative values for the planning based approach mark that the program failed to produce a solution with the memory limit of 1,000MB. The redundant steps could be removed from the planning-based game solver too, by maintaining a set of previously applied steps and verifying that the new step to be added was not applied before. However, this approach is not feasible in practice, as it increases the memory requirements of the program.

Note that the comparison in Fig. 2 is not entirely accurate: while the planning-based approach finds just one solution, but the shortest one, the model-based approach was set to find all solutions, but cannot tell, which one is the shortest. Since we learned from Anderson and Feil [8] that any solvable 5×5 puzzle has exactly 4 solutions, finding the shortest one based on a model-based approach is a simple task.

III. GENERATING LIGHTS OUT! GAMES USING FIRST ORDER LOGIC

Our goal here is to automatically generate puzzles that have a solution by reasoning on finite models of first order logic theories. We designed three solutions: (i) using a LIGHTS OUT! game solver in FOL, (ii) using linear algebra and a

TABLE I: Five games with one solution for each

G_1	G_2	G_3	G_4	G_5
$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$
$[(0,0) \quad (0,4)]$	$\begin{bmatrix} (0,0) & (0,4) \\ (2,2) & (4,3) \\ (4,4) & \end{bmatrix}$	$\begin{bmatrix} (0,0) & (0,1) \\ (0,3) & (0,4) \\ (4,0) & (4,4) \end{bmatrix}$	$\begin{bmatrix} (0,0) & (0,4) \\ (1,1) & (1,2) \\ (1,3) & (2,0) \\ (2,4) & (4,2) \end{bmatrix}$	$\begin{bmatrix} (0,3) & (0,4) & (1,0) & (1,1) \\ (1,3) & (1,4) & (2,1) & (2,2) \\ (2,4) & (4,1) & (3,2) & (3,3) \\ (4,0) & (4,2) & (4,3) & \end{bmatrix}$

TABLE II: Finding the minimal number of moves

$Model_1$	$Model_2$	$Model_3$	$Model_4$
$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$
10 moves	10 moves	2 moves	18 moves

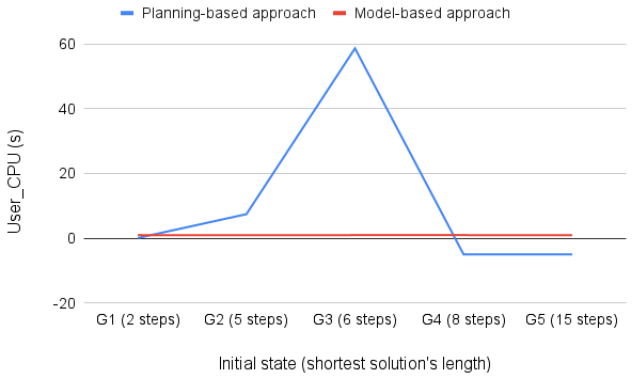


Fig. 2: Comparing the *User CPU* time for solving the game Note that the negative User_CPU(s) values show that the execution was unsuccessful with a memory limit of 1000MB.

model generator and (iii) improving the linear algebra-based method by decreasing the domain size.

A. Using puzzle solving algorithm

First, one can generate LIGHTS OUT! games by using the above model-based solving method. One change is to let initial game state being unspecified. To do so, what we need to do is only to remove the propositions about the $on(x, y)$ function, and specify only that there must be at least one light bulb which is on in the initial state:

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

Some observations follow. First, this method does unnecessary things: note that our goal was just to generate a solvable puzzle, not that of solving it. Second, the same puzzle is generated multiple times. The models generated by Mace4 include the solution to the puzzle also, and each solvable puzzle has multiple (that is exactly 4) solutions [8]. Hence for each n generated puzzles/models, we would get only $\frac{n}{4}$ different solvable puzzles, and it would be an additional task to identify, which models represent the same puzzles.

B. Using linear algebra and model generator

The second solution is based on the work of Anderson and Feil [8]. Anderson and Feil have introduced the following three theorems on: (1) an equivalent criterion for the solvability of a game state; (2) the number of solutions of a game state, and (3) the format of the solutions. Let the initial state of the game represented by the column vector B, where $B[i] = 0$ iff the light bulb in row $i/5$ and column $i\%5$ is off, and $B[i] = 1$ for light on, with

$$\vec{B} = On(0), On(1), On(2), \ldots On(24) \tag{4}$$

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

Theorem 1: B is a solvable game state $\iff$ $B \perp N_1$ & $B \perp N_2$ where

$$\vec{N_1} = (0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0) \tag{5}$$

$$\vec{N_2} = (1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1) \tag{6}$$

Theorem 2: Each solvable game state has exactly 4 solutions.

Theorem 3: If $\vec{X} = (Switch(0) \quad Switch(1) \quad Switch(2) \quad \ldots \quad Switch(24))$ is a solution for game state B, then the 4 solutions are

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

To apply these theorems to the task, we ask Mace4 to fill the vector On, representing the initial state of a solvable

game, with 0's and 1's (at least 1 value of 1), such that On is perpendicular on N_1 and N_2. We can easily verify the perpendicularity by computing the dot products:

$$\vec{On} \perp \vec{N1} \leftrightarrow \vec{On} \cdot \vec{N1} = 0 \quad (8)$$
$$\vec{On} \perp \vec{N2} \leftrightarrow \vec{On} \cdot \vec{N2} = 0 \quad (9)$$

In Mace4, using the arithmetic library with a domain size of 26, we can define N_1 and N_2 as functions, e.g., $N_1 : [0..25] \rightarrow \{0, 1\}$, e.g. $N_1(0) = 1 \wedge N_1(1) = 0 \wedge N_1(2) = 1 \wedge \ldots \wedge N_1(25) = 0$. Note that N_1, N_2 and On are indexed from 0, to allow an easier description of the following relations, but the game state is in fact represented by $On : [1..25]$.

We also specify the relations for the dot products:

$$\begin{aligned}(x > 0 \wedge y + 1 = x \wedge Z = DotProd_1(y) \wedge \\ O = On(x) \wedge N = N_1(x)) \rightarrow \\ DotProd_1(x) = Xor(Z, And(O, N)). \quad (10)\\ DotProd_1(0) = 0. \quad (11)\\ (x > 0 \wedge y + 1 = x \wedge Z = DotProd_2(y) \\ \wedge O = On(x) \wedge N = N_2(x)) \rightarrow \\ DotProd_2(x) = Xor(Z, And(O, N)). \quad (12)\\ DotProd_2(0) = 0. \quad (13)\end{aligned}$$

so the conditions for perpendicularity can be expressed as $DotProd_1(25) = 0$ and $DotProd_2(25) = 0$.

Finally, Mace4 has to search for the unknown values of the function On, for which two conditions hold: (1) $On(0) = 0$ and (2) $\exists x\ On(x) = 1$.

The advantage of model based approach over planning is that searching occurs only on the values representing the states of the light bulbs in the initial state, and not for a sequential solution of the puzzle, as the planning-based approach does. However, the generator is slow because it needs to generate all the models. If one tries to run the algebra-based code with a smaller domain size and smaller vectors, then the program runs fairly quickly and produces unique results. Also, if one takes a look at the model generated by Mace4, it can be noticed that because of using domain size 26, all the helper functions are unnecessarily large (the binary relations And and Xor have 25*25 = 625 elements). Defining all the unnecessary values, even if their value can be easily deduced and no backtracking is needed, takes time for Mace4. This leads to a way for improving the model based approach by *reducing the domain size*.

C. Improving the linear-algebra based method

To reduce the domain size, we reorganize the vectors $On(x)$, $N_1(x)$, $N_2(x)$, $DotProduct(x)$ with $1 \leq X \leq 25$ into 5×5 matrices $On(x, y)$, $N_1(x, y)$, $N_2(x, y)$ and $DotProd(x, y)$ with $0 \leq X,\ y \leq 4$. Thus, we can use a domain size of 5 instead of 26. The vectors N_1 and N_2 will be represented as binary functions:

$$N_1(x,y) = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} N_2(x,y) = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

The relations for computing the dot products become

$$\begin{aligned}(y \geq 1 \wedge yn + 1 = y \wedge d = DotProd_1(x, yn) \wedge \\ o = On(x, y) \wedge n = N_1(x, y)) \rightarrow \\ DotProd_1(x, y) = Xor(d, And(o, n)).\\ (x \geq 1 \wedge xn + 1 = x \wedge d = DotProd_1(xn, 4) \wedge \\ o = On(x, 0) \wedge n = N_1(x, 0)) \rightarrow \\ DotProd_1(x, 0) = Xor(d, And(o, n)).\\ 0 = On(0, 0) \wedge n = N_1(0, 0) \rightarrow \\ DotProd_1(0, 0) = And(O, N).\\ (y \geq 1 \wedge yn + 1 = y \wedge d = DotProd_2(x, yn) \wedge \\ o = On(x, y) \wedge n = N_2(x, y)) \rightarrow \\ DotProd_2(x, y) = Xor(d, And(o, n)).\\ (x \geq 1 \wedge xn + 1 = x \wedge d = DotProd_2(xn, 4) \wedge \\ O = On(x, 0) \wedge n = N_2(X, 0)) \rightarrow \\ DotProd_2(x, 0) = Xor(d, And(o, n)).\\ O = On(0, 0) \wedge N = N_2(0, 0) \rightarrow \\ DotProduct2(0, 0) = And(O, N).\end{aligned}$$

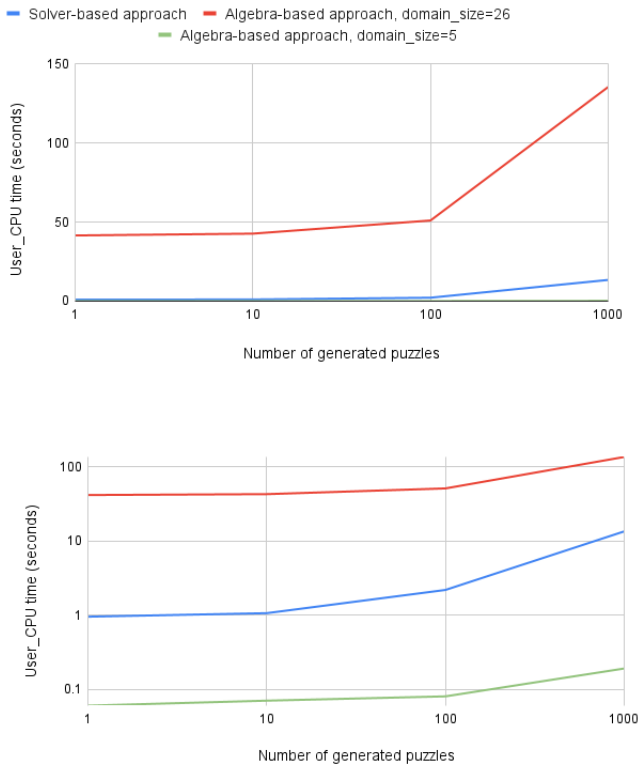
where Mace4 has to find the values of $On(x, y)$ with three constraints:

$$On(x, y) = 1 \vee On(x, y) = 0 \quad (14)$$
$$\exists x\ \exists y\ On(x, y) = 1 \quad (15)$$
$$DotProd_1(4, 4) = 0 \wedge DotProd_2(4, 4) = 0 \quad (16)$$

With this simple improvement, the decrease in execution time is more than significant: generating one model takes just a fraction of a second. Figures 3a and 3b bear out the differences in the performance of the Mace4 based on the three approaches, based on the User CPU time. Here Fig. 3b uses a logarithmic scale. Note that for finding n solvable puzzles, from Mace4 with the algebra-based approaches, n models were requested, but with the approach based on a game solver, $4 * n$ models were requested.

The improved algebra-based approach outperforms all the other ones: it can generate 1000 solvable puzzles in 0.19 seconds, which makes it well-suited for being integrated in a LIGHTS OUT! game with a graphical user interface. For example, it can be used to generate, in real-time, a large number of different puzzle configurations for each new game.

Maybe experiments with generating random bitstrings of length 125, keeping the ones that satisfy the two checks for perpendicularity. Each test succeeds with probability 0.5, so this must be as fast as you could possibly want. Why would



(b) logarithmic scale

Fig. 3: Comparing puzzle generators based on UserCPU time

we use search as in Section 3, rather than a simple generate-randomly-and-test? (I fill that is what exactly MAce4 does - gnerates bitstrings of length 125 (5x5), keeping the ones that satisfy the two checks for perpendicularity).

We introduced here two approaches in terms of generating solvable instances of the LIGHTS OUT! puzzle. The first relies on a modified version of the model-based algorithm to solving an existing instance, while for the second one an optimization is presented to an existing approach based on linear algebra. Experimental results shown that the optimized version of the algebra-based algorithm yields significantly better performance compared to both its original counterpart as well as the model-based approach.

IV. REUSING KNOWLEDGE FOR LIGHTS OUT! VARIANTS

There are many variations of the original LIGHTS OUT! game, for example (i) changing the size of the board; (ii) the pattern of the moves (diagonal, the neighbours are affected but not the node itself); (iii) generalised to arbitrary graphs instead of a grid; or (iv) the number of possible states for each cell, i.e., instead of toggling between on and off, we may consider a cycle $on \rightarrow color_i \rightarrow on$. We formalised these variants by reusing parts of the FOL-theory for the LIGHTS OUT! puzzle.

Consider the formalisation for patterns of moves other than a vertical cross, e.g. the diagonal cross, torus, or the knight's move from chess, or even more difficult when changing lights that are not adjacent but further away then light chasing. For instance, the diagonal-pattern for a 5×5 is formalised with:

$$\begin{aligned} &5\_diag(x,y) \leftrightarrow \\ &(x \neq 5 \wedge x \neq 10 \wedge x \neq 15 \wedge x \neq 20 \wedge x \neq 25 \wedge \\ &x+1 == y) \vee \\ &x == y \vee \\ &x+5 == y \vee \\ &(y \neq 5 \wedge y \neq 10 \wedge y \neq 15 \wedge y \neq 20 \wedge y \neq 25 \wedge \\ &y+1 == x) \vee \\ &y+5 == x \end{aligned} \tag{17}$$

For a general $n \times n$ grid the affected cells after a light is toggled (using the diagonal pattern) would be given by:

$$\begin{aligned} &n\_diag(x,y) \leftrightarrow \\ &(x \neq n \wedge x \neq 2n \wedge .....x \neq n^2 \wedge x+1 == y) \vee \\ &x == y \vee \\ &x+n == y \vee \\ &(y \neq n \wedge y \neq 2n \wedge ...y \neq n^2 \wedge y+1 == x) \vee \\ &y+n == x \end{aligned}$$

The number of states assigned to a cell could be extended as well. In this line, the Lights Out 2000 variant came with 3 states assigned to each cell, which corresponded to 3 different colors. On each button press, the cell changes from one state (colour) to the next. The effect of changing the state of a light affects not only the clicked light button but also its direct neighbours, as defined by the current pattern (e.g. cross). The total number of switches of a given button together with the number of switches of its neighbours account to the final state of the button, which should be 0, corresponding to the final (off) state.

For formalising the effect of these switches on each button's state in FOL, instead of using the *xor* operation, we use the addition, for operands outside the range of modulo 2: Let the functions $b(x,y)$ representing the initial state of the cell, and $c(x,y)$ the number of times a cell's state is changed. Let $ds = domain\_size - 1$ the size of the extended grid and $noStates = 3$. After applying a number of moves (i.e. switches) on the current and neighbour cells, the final state of the current cell should be the state 0. That means, if the cell $b(x,y)$ is not in the correct state, it should be changed either directly or through its neighbours:

$$\begin{aligned} &(x > 0 \wedge x < ds \wedge y > 0 \wedge y < ds \wedge xp = x-1 \wedge \\ &yp = y-1 \wedge xs = x+1 \wedge ys = y+1) \rightarrow \\ &(b(x,y) + c(x,y) + c(xp,y) + c(x,yp) + c(xs,y) + c(x,ys)) \\ &mod\ noStates = 0 \end{aligned}$$

Here xs and ys represent the successor of values x and y, while xp and yp their predecessor values.

V. Discussion and related work

The LIGHTS OUT! game can be seen as a recreational exercise. Yet, there is more than recreation, since LIGHTS OUT! can be used to illustrate logic concepts or theorems on graph structures. The difficulty of solving such a LIGHTS OUT! game increases by the dimensions of the given board configuration, or the pattern of lights that can be toggled with a single move. Even for the 5×5 grid, human agents face difficulties to find a solution, especially an optimal one, which does not contain redundant moves (switching the same light an even number of times).

To solve the game, various algorithms can be applied, including the *Light chasing* method [9]. Starting with an initial state of cells and a set of rules about how these cells interact with their neighbors, a simulation of the change in the cells states can be run over time. The Light chasing method, similar to Gaussian elimination, guarantees to solve the problem, if a solution exists, but it might require many redundant steps. It successively scans each row, starting with the second one, and clicks on every cell which has a light above it, on the previous row. The last row is handled separately, using a lookup table with some predefined light patterns in order to identify which lights should be toggled in the first row. After that the initial algorithm is applied again starting from the second row, until a solution is found. Instead of such procedural approach, we described here various declarative approaches to solve the game and also to automatically generate new LIGHTS OUT! configurations.

Logic games and puzzles, such as the LIGHTS OUT!, can be used to help visualize and better understand some logic aspects, such as graph theory, planning, search algorithms, or finding repetitive patterns which lead to a solution (even if not an optimal one) as in case of the *Light chasing* method [10].

A similar game to the LIGHTS OUT!is the *Khalou* puzzle, which consists of a 4×4 array of stones, each with a white and a black side. During one move, the stones on a complete row and column are flipped, instead of the cross-diagonal pattern as in LIGHTS OUT!. For the 4×4 layout, the upper bound for the number of moves in which the *Khalou* puzzle can be solved is considered to be 5 steps. Hopkins has introduced an additional constraint - a timer - which requires to find a maximum 5-moves solution in the given time limit [11]. By adding different edges to the $n \times n$ grid graph, Davis et al. have listed some extensions including the *cylindrical* LIGHTS OUT!, *toroidal* LIGHTS OUT!, *Mobius strip* LIGHTS OUT!, *Klein bottle* LIGHTS OUT!, or *real projective plane* LIGHTS OUT! [2]. These extensions can be solved by reusing FOL axioms for the classical LIGHTS OUT!, and adding the new constraints for each variant.

As argued by Meyer, such logic puzzles can be an important asset in the education as well, contributing to the development of logic thinking, reasoning and exploring the problem solving-process, all these in an interactive way [12], [13]. From the mathematical perspective, various theorems and properties can be derived when modeling the game using a graph representation and in the domain of linear algebra (some such theorems are listed by Davis et al. [2]), while taking into consideration certain game invariants [14]. Using the adjacency matrix for representing the graph and by applying the rules of linear algebra [2], [14], one can determine, for example if the given matrix is reflexive, symmetric or invertible, which could help to derive theorems and proofs, later used in finding more optimal solutions for the LIGHTS OUT! game [15].

VI. Conclusion

The paper looks at the well-known LIGHTS OUT! game and two associated computational problems: (1) how to solve an instance (or determine that no solution is possible), and (2) how to generate solvable instances.

We presented here various declarative methods to solve and automatically generate LIGHTS OUT! games. The first method is based on a plain implementation of a solver by modelling the game as a planning task in FOL. The only advantage here is the user have access to the proof used to build the plan. The second method incorporates clever observations such as: (i) lights that are on must be toggled an odd number of times to be turned off; (ii) lights that are off must be toggled an even number of times for them to remain off; (iii) given a solution, the order in which the lights are pressed does not matter. The declarative solutions above allow the software engineer to easily extend the solvers to various variations of the LIGHTS OUT!-game.

Based on the running experiments, we could draw the following conclusions: First, LIGHTS OUT! games can be solved both by planning in FOL, with Prover9's production mode, and through model finding methods. Second, while Prover9's production rules can be used to generate plans by proving theorem in FOL, if the order of the steps is irrelevant, using model finders is more efficient. The difference in solving efficiency between the "planner approach" and the "model approach" is expected, since the former does not exploit the various redundancies and symmetries (e.g. duplicate moves never being part of an optimal solution, or order of moves being irrelevant for the outcome). Third, solvable LIGHTS OUT! games can be generated based on game solvers and based on model finders relying on linear algebra as well. Fourth, 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. Using $n \times n$ matrices instead of $1 \times (n^2)$ array reduces the execution time.

One line is to translate the problem into propositional logic, since it will generate formulas that make heavy use of xor and conjunction. It would be good to know how a state-of-the art SAT solver performs as a solver for this problem, or perhaps a solver like CryptoMiniSat, which will take xor form as input. Knowing that the order of steps in a solution is irrelevant and this problem is all about parity, one just need to identify the relevant set of bulbs.

REFERENCES

[1] E. F. Meyer, N. Falkner, R. Sooriamurthi, and Z. Michalewicz, *Guide to teaching puzzle-based learning*. Springer, 2014.

[2] T. Davis, L. Grimley, K. Ince, G. Karaali, B. Kostadinov, and R. Soto, "From puzzles to proof-writing: Exploring rich mathematical ideas through mechanical puzzles," *Teaching Mathematics Through Games*, vol. 65, p. 97, 2021.

[3] M. Danesi, *Ahmes' legacy: Puzzles and the mathematical mind*. Springer, 2018.

[4] ——, *An Anthropology of Puzzles: The Role of Puzzles in the Origins and Evolution of Mind and Culture*. Routledge, 2020.

[5] G. Sutcliffe, "The TPTP problem library and associated infrastructure: from CNF to TH0, TPTP v6. 4.0," *Journal of Automated Reasoning*, pp. 1–20, 2017.

[6] A. Groza, *Modelling Puzzles in First Order Logic*. Springer Cham, 2021.

[7] W. McCune, "Prover9 and Mace4," 2005.

[8] M. Anderson and T. Feil, "Turning lights out with linear algebra," *Mathematics Magazine*, vol. 71, no. 4, pp. 300–303, 1998.

[9] C. D. Leach, "Chasing the lights in lights out," *Mathematics Magazine*, vol. 90, no. 2, pp. 126–133, 2017. [Online]. Available: https://doi.org/10.4169/math.mag.90.2.126

[10] M. A. Madsen, "Lights out: Solutions using linear algebra," *Summation*, vol. 3, pp. 36–40, 2010.

[11] B. Hopkins, "Can You Win Khalou in Five Moves?" in *The Mathematics of Various Entertaining Subjects*. Princeton University Press, 2019, pp. 204–212.

[12] R. S. Meyer, "The game of Lights out," 2013.

[13] A. Lisitsa, "Revisiting mu-puzzle. a case study in finite countermodels verification," in *International Conference on Reachability Problems*. Springer, 2018, pp. 75–86.

[14] A. Berman, F. Borer, and N. Hungerbühler, "Lights out on graphs," *Mathematische Semesterberichte*, pp. 1–19, 2021.

[15] R. Fleischer and J. Yu, "A survey of the game "Lights Out!"," in *Space-efficient data structures, streams, and algorithms*. Springer, 2013, pp. 176–198.