K-turns SAT Encoding for “Lights Out” in Java

CS 1800 Logic and Computation

[redacted]

Abstract

In this paper, we focus on building a SAT encoding for the game of Lights Out, a single-player game in which the player must “turn off” all tiles on a board. We will first describe the gameplay, specifically how moves can be made and how they affect the board. Then, we will discuss a traditional strategy to mathematically solve the game using linear algebra and then suggest our approach to obtaining a winning solution via a SAT solver. In our solution, we integrate the Z3 SAT solver library in a program written in Java that prompts a game configuration described with two parameters, one of which is *k*, the number of moves to solve the game in. If a solution exists, the program outputs a visual of the solution. Otherwise, it notes that there is no such solution. See [redacted] for the source code.

Table of Contents

I. Introduction

There exists a wide breadth of single-player games with SAT encodings developed by SAT enthusiasts and researchers. The foray of the SAT community into game encodings around the early 2000s commenced with a focus on classic games such as Sudoku and Eight Queens. These encoding problems are now known as 'Constraint satisfaction problems' (Wikipedia), or encodings that are built off of constraints based on the structure of the games and their rules. A wave of constraint-based encodings for a myriad of other similar games soon followed such that, even games offered primarily on mobile apps now have their own encodings.

In this paper, we offer what we believe to be one of the first encodings for Lights Out, a single-player game played primarily through internet game sites, in which the solution lies in the set of moves made by the player. To solve this game, we enumerate a set of constraints necessary to winning a Lights Out game and provide them to the Z3 SAT Solver Java library to evaluate. We will employ the SAT solver in a program that first prompts user input and outputs according to whether or not it's possible to win the game specified by the user input. We determined that we would be successful in this project if we could develop a SAT encoding for a game of the given inputs n and k that, when used with a SAT solver, could tell the player if such a game was winnable.

II. Background

Lights Out Rules and Gameplay

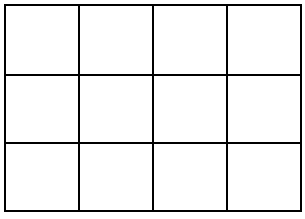

Figure 1 A. 3x4 Two-State Lights Out board with all tiles “on”

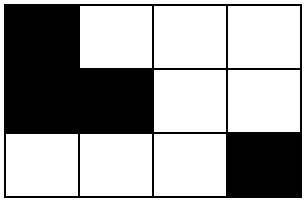

Figure 2. A 3x4 Two-State Lights Out board with some tiles “off”

We will now explain the configuration of a Lights Out game, how moves work, and how to win a game. A game of Lights Out consists of a board with *m* rows and *n* columns of tiles where *m* and *n* are positive integers. Tiles have a *p* number of states numbered [1,*p*], in which we consider a tile that has reached the *p*-th state to have been “turned off”. Tiles always iterate through states successively from the first state seen in the initial board-state to the *p*-th state. If a tile in the state *p* must switch states again, the tile goes back to state 1. In this paper, we focus on games with only two states in which we will denote the two states as a tile either turned “on " or “off”. A move can be made on a certain tile by clicking on the tile, henceforth, we will interchangeably refer to moves as clicks within this paper. For example, clicking a tile of two states initially turned “on" twice consecutively will leave it “on”.

The game will start with all tiles in the “on” state, as depicted in Figure 1, in which the objective of the game is to turn “off” all tiles. Figure 2 depicts a board with some tiles turned “off”. Clicking a tile once will toggle its state, as well as the states of tiles adjacent to the clicked one. As a demonstration, if we click the tile at row 2, column 3 in Figure 2, the resulting board will be as depicted in Figure 3, where the white x-mark denotes the location of the clicked tile.

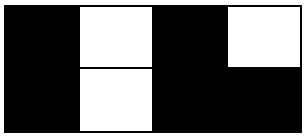

Figure 3. 3x4 Two-State Lights Out Board After a Move at the Marked Tile

The tile to the immediate left of the marked tile is flipped from "off" to "on". The other three adjacent tiles and the marked tile flip from "on" to "off". Note that the number of adjacent tiles varies according to the location of the clicked tile. For example, while the marked tile has four adjacent tiles, a tile at the corner will only have two.

SAT Encoding for Peg Solitaire

We will now introduce an encoding for Peg Solitaire described by Do, Chuong of Stanford that we adapted to our Lights Out encoding (Do). With Lights Out, all final board states are the same because the board must have all tiles turned "off" by the end of the game. This is similar to Peg Solitaire in which a player can only win a game when all spaces on the board but one are empty. Thus, encoding constraints for Peg Solitaire must rely on validating a sequence of moves using board-states resulting per move. Boards consist of spots represented by boolean values where *true* means the spot has a peg and *false* means the spot is empty.

For two board-states *p* representing the board before a move and *q* representing the state after a move, we consider all possible moves on *p*, valid and invalid. We check if each possible move is valid from *p* to *q* according to the following four constraints: A move is valid when the move was made from a *true* spot, the jumped over spot is *false*, the spot jumped to is *true*, and all other spots on the board are the same in *p* and *q*. We can find the single valid move for a *p,q* pair by OR-ing all possible moves, and checking their validity in the context of *p,q*. Then, we can proceed from board-state to board-state until we reach the *(n-2)th* board-state where *n* is the

number of spots on the board. By the *(n-2)th* board-state, all but one spot will be *false,* or empty, and we win the game. Thus, we have a CNF encoding of Peg Solitaire.

The Z3 Satisfiability Solver

Our solution for the Lights Out game makes use of the Z3 Satisfiability Solver[1]. If the Z3 SAT Solver finds a given set of constraints satisfiable, it returns the set of boolean values that satisfies the constraints. Otherwise, the solver denotes that the given expression is unsatisfiable. Z3 represents boolean variable literals as the datatype *BoolExpr*, which we use to represent tiles of a Lights Out game.

III. Methodology

Our implementation of the Lights Out solver utilizes a *k-turns* algorithm in which we flip the state of each tile for successive turn according to the tile's state in the previous turn (except turn 0, the initial board-state). Coming up with an encoding was the most strenuous aspect of this project, particularly the parameterization of the problem. We initially found our particular parameterization not the most intuitive, so understanding how the encoding would look under this parameter was the most difficult part of encoding the game. In this section, thanks to the gratuitous help of Professor Hemann, we will discuss the steps we took to reach a functional *k-turns* implementation.

Preconditions

For this SAT encoding of Lights Out, we will focus on square boards as opposed to boards of differing width and height dimensions, such as the boards featured in Figures 1, 2, and 3. All other rules of Two-State Lights Out apply.

[1] https://github.com/Z3Prover/z3

Parameters

This SAT encoding is dependent on two parameters the player has to provide to our program, the first being *n,* the width and height of a Lights Out board, and the second being *k*, the number of moves in which to win the game.

Objectives and Outputs

If it's possible to win the specified Lights Out board in *k* moves, our program will output a 2D array of 0's and 1's where 1's represent the set of tiles the player must click once, in no particular order, to turn all tiles "off" and 0's represent tiles that the player doesn't need to click to win the game. Otherwise, the program prints "Unable to solve board in *k* moves." For example, if a user gives the program $n = 3$ and $k = 5$ for a 3×3 Two-State Lights Out game solved in 5 moves, the program prints the following 2D array:

1 0 1

0 1 0

1 0 1

Figure 4. Solution for a 3x3 Two-State Lights Out board with $k = 5$

If the user gives a $k < 5$, the program outputs "Unable to solve board in 4 moves."

Setup

We represent a Lights Out board as a 2D BoolExpr array of size $[n][n]$. Since our implementation must keep track of the board state per *k* move made, we declare a 3D BoolExpr array named *grid* of size $[k][n][n]$, or an array of *k* boards:

$$BoolExpr[][][]\ grid\ =\ new\ BoolExpr[k][n][n]\ .$$

We also need to implement an array of transitional arrays, *move*. Each BoolExpr in *move* represents the tiles that can be flipped for each of the *k* moves. Since every tile on the board can be flipped, the *move* array is identical to the *grid* array:

$$BoolExpr[][][]\ move\ =\ new\ BoolExpr[k][n][n]\,.$$

If a $m_{turn,row,col}$ is true, then the flipped tile of the turn is $p_{turn,row,col}$.

As aforementioned, the starting state of all tiles is "on", for which we will represent as *false*. Hence, all tiles in the board $grid[0]$ will be set to *false*. For tiles in the following $k - 1$ boards of *grid*, we represent each as the Boolean constant $p_{turn,\,row,\,col}$, where *turn* represents the number of moves already made and *row* and *col* represents the tile's position within its board. For example, we represent the tile (0,2) on a board after one move with the constant $p_{1,\,0,\,2}$[2]. Or, in technical terms, *turn* denotes the position of a board within the *grid* array such that a board representing one move made is the board at *grid*[1]. We will represent every tile in *move* with a similar method, except we represent each tile in the board *move*[0] as the constant $m_{0,\,row,\,col}$ instead of *false*.

Constraints

Our encoding involves four constraints. First, for each move, we must keep track of the adjacent tiles that change states as a result of a move. To demonstrate, if we click tile (0,0) on the initial board-state, then the adjacent tiles (0,1) and (1,0) also switch states. This example can be alternatively expressed as:

$$m_{0,0,0} \Rightarrow ((p_{1,0,1} \neq p_{0,0,1}) \wedge (p_{1,1,0} \neq p_{0,1,0}))$$

[2] In the actual implementation, this constant would be displayed as "p102". The usage of subscript and commas is to increase the clarity of such representation.

However, instead of ≠, we use XOR to ensure that the state of the tile in the next turn is different from its current state. For example:

$$p_{1,0,0} \otimes p_{0,0,0}$$

Thus, to represent multiple tiles flipping at once as the result of a single move, simply AND the XOR expression of each flipped tile:

$$(p_{0,0,0} \otimes p_{1,0,0}) \wedge (p_{1,0,1} \otimes p_{0,0,1}) \wedge (p_{1,0,1} \otimes p_{0,0,1}).$$

Second, to ensure that no other tiles changed states, for all tiles excluding (0,0), (0,1), and (1,0), equate the current and next states of the tiles:

$$(p_{1,1,1} \Leftrightarrow p_{0,1,1}) \wedge (p_{1,1,2} \Leftrightarrow p_{0,1,2}) \wedge \ldots$$

Then, AND the two above constraints such that the resulting expression represents all tiles on the board. For example, clicking tile (0,0) on a 2×2 board will result in the below constraint:

$$m_{0,0,0} \Rightarrow (p_{0,0,0} \otimes p_{1,0,0}) \wedge (p_{1,0,1} \otimes p_{0,0,1}) \wedge (p_{1,0,1} \otimes p_{0,0,1}) \wedge (p_{1,1,1} \Leftrightarrow p_{0,1,1}).$$

Note that the number of adjacent tiles differs depending on the position of the *move* tile, so distinguishing the constraints for corner and edge tiles from center tiles are necessary.

Third, we will only allow one move per turn by XOR-ing each move such that only allow one board in the *move* array is true for each turn:

$$m_{0,0,0} \otimes m_{0,0,1} \otimes m_{0,1,0} \otimes m_{0,1,1},$$

and so on for all other turns until after turn *k*. We will discuss the fourth constraint in the following section.

SAT Solver and Evaluation

Once we provide the Z3 Solver our constructed constraints for all possible games of *k* moves, we apply our fourth constraint which finds a winning solution grid[i], where *i* is any valid

index of *grid,* such that all tiles in its *k*-th board-state are true. If this constraint is not satisfiable, our Lights Out solver will output the failing message stated in *Objectives and Outputs*. Otherwise, our implementation will access the combination of *move* tile values returned by Z3 and count the number of times each tile evaluates to *true*. The program then assigns values to all spots in a 2D array $result[n][n]$ such that the given values denote the winning set of moves the user must play to win the game. Each tile will assume the value 1 if its respective *move* tile evaluates to *true* an odd number of times and 0 otherwise. To solve the game themselves, the user may click on all tiles in a game, in no particular order, whose respective location in the *result* array contains a 1.

IV. Conclusion

In this paper, we have discussed our successful implementation of solving the Lights Out game using the Z3 SAT Solver Java library. We revealed that our program takes in two inputs, *n* representing the dimension of a square game-board, and *k* representing the number of moves in which to win the game. Our program then generates constraints that it provides to the Z3 Solver to evaluate and determine a winning solution to the user-defined game. According to our pre-defined measure of success stated in *Introduction*, we were successful in our project. Our implementation outputs the correct solution for the provided number of turns, as supported by solutions provided by other Lights Out solvers (Liste).

However, our project parameterized the game using the number of moves in which to solve the game, which we chose to do to simplify the problem and allow us to complete the project by the deadline. Furthermore, our implementation has a time complexity of $O(n^2 \times k)$ such that, for all $k > n$, our implementation becomes less efficient than traditional Lights Out solvers that run in time $O(n^3)$. Lastly, for boards larger than 5×5, the program's feedback time

exceeds 2 seconds regardless of the input for *k*. Therefore, future improvements upon our program would focus on increasing efficiency and eliminating the *k* parameter to output the optimal minimal solution. As previously stated, our encoding should be one of the first developed specifically for Lights Out. While we don't believe the results of this paper pose any implications for unsolved SAT problems involved in research, our results support the idea that amateur SAT enthusiasts of any level can encode almost any polynomial-time decision problems in the world with reference to existing encodings.

Bibliography

Anderson, Marlow, and Feil, Todd. “Turning Lights Out with Linear Algebra.” *Mathematics Magazine*, Oct. 1998, pp. 300–303.

Do, Chuong. “Satisfiability and Peg Solitaire.” *Satisfiability and Peg Solitaire*, https://ai.stanford.edu/~chuongdo/satpage/index.html. Accessed 8 Nov. 2020.

Liste, Rafael. “Lights Out (Games with solutions)”. *Geogebra*, 14 June 2018, https://www.geogebra.org/m/JexnDJpt. Accessed 8 Nov. 2020.

“Solving the 3×3 Lights Out .” *Solving 3 x 3 Lights Out*, people.math.carleton.ca/~kcheung/math/notes/MATH1107/wk04/04_solving_3x3_lights_out.html.

Wikipedia contributors. "Constraint satisfaction problem." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 15 Oct. 2020. Web. 9 Nov. 2020.