Kyle, Sean, Tristan

Prof. Fitzsimmons

Artificial Intelligence

4 November 2022

Light Up Puzzle Problem

In this project, we wrote a Python program to solve a "Light Up" puzzle using a SAT solver. Each puzzle contains a grid with obstacles and numbers. When lightbulbs are placed in empty spaces on the puzzle, they shine a light in each of the four directions on the grid until the end of the grid, an obstacle, or a number is reached. To solve the puzzle, one must place lights so that each empty space on the grid is lit up. In addition, no two lights can light one another, and each number on the grid must have exactly that many lights directly adjacent to it.

Our three sets of rules for this puzzle were "Row/Column rules," "Light rules," and "Number rules." For the Row/Column rules, no two spaces in a section (either a row or column bounded by a wall, number, or obstacle on both sides) could both be lit up. The CNF form looks like this: $(\neg A \vee \neg B) \wedge (\neg A \vee \neg C)$…, where in each pair, only one space could be true, otherwise the whole expression would become false. For the Light rules, every empty space (".") had to be lit by a light. With barriers and numbers blocking light, this means each space must be in direct line of sight with a light. In CNF, this will look like $(A \vee B \vee C…)$, where only one light in each spaces sightline must be true for the expression to be true. Finally, for Number rules, for each number (0-4), the adjacent squares filled with lights must equal the number. For each number representing the amount of adjacent lights in the solution, we wrote down the different possibilities for each combination of adjacent lights in propositional logic form. From there, we converted each logic statement into CNF notation giving us the rules we needed for certain cases of the puzzle. For example, for the “1” case with three adjacent neighbors that could be lit up, we had $A \to \neg B \wedge \neg C \Rightarrow (B \vee \neg A) \wedge (\neg C \vee \neg A)$.

In order to generate solution models for each puzzle, we implemented four functions called neighbors(), rowcol_rules(), light_rules(), and number_rules(). The neighbors function allowed us to generate valid neighbors for each gridvariable in order to avoid going out of bounds. We then used these valid neighbor values in the number_rules() function which ensured that each number constraint is satisfied. The light_rules() function ensures that all empty spaces are illuminated, by checking all spaces in a “plus” shape around the current square, until an obstacle, number, or wall is hit. Knowing the amount and locations of valid positions surrounding each number was important because the CNF formula is different based on the number of valid positions. For example, the ‘3’ case is simple when there are only three valid neighbors because it there is only one possibility. When there are four valid neighbors, however, there are 4 possibilities. In the rowcol_rules() function, we implemented the CNF constraints that ensured no two lights shine on one another, similar to checking whether or not two Rooks were in the same row/column in the Rooks.py code.

We experienced some challenges implementing each of the functions. Initially in light_rules() we only checked spaces to the right and down, and had duplicate numbers in our “plus”. This made error-checking difficult, but it wasn’t an extremely hard fix to make. The first row_col() function that we made wouldn’t allow for more than one light to be in the same row or column regardless of an obstacle or number being between them. To fix this issue, we created variables *prevBarrier* and *lastBarrier* to keep track of the barriers for each row/column. Then each row/col that is broken up by a barrier is treated as its own row/col allowing for multiple lights to be on a row or column if a barrier is between them.