Tom and Jack: Irrational Agents Write-up

<u>**Description of Problem:**</u>

In this problem, the name of a puzzle file is provided in the command line. This puzzle contains a rectangular board with blank spaces denoted by '.', obstacles represented by 'X', and numerical obstacles from 0 to 4. The object of this game is to light up every square on the board. No two lights can be in the same row or column unless they have at least one obstacle between them. When a number appears in the puzzle, there must be that many lights as direct neighbors to that board position. When all three of these constraints are met, the puzzle will be solved.

<u>**SAT Encoding:**</u>

For each of the constraints mentioned in the problem description, there is an associated SAT encoding. For clarity, we will split this section into descriptions of each constraint. A tile having a positive or true value indicates a light on that tile, while a "not" or negative value indicates no light on that tile.

<u>*Rooks Constraint:*</u>

For a given tile 'A' and another tile 'B' that is horizontally or vertically in line-of-sight to tile 'A' (i.e. no obstacles between them), A and B cannot both have lights on them at the same time. In boolean logic, one would write this as $A \Rightarrow \neg B$, or $\neg A \vee \neg B$ in conjunctive normal form (CNF).

<u>*All tiles lit constraint:*</u>

For every tile on the board that is not an obstacle, the set of tiles including the current tile and all of the tiles both horizontally and vertically in line-of-sight of the current tile needs to include a light. For a current tile A and set of visible tiles {B,C,D,E,F}, the rule is $A \vee B \vee C \vee D \vee E \vee F$.

<u>*Obstacle constraint:*</u>

Because we know that an obstacle, either a number or 'X', cannot have a light on that tile, we ran through the board, checking for all obstacles, and added the constraint $\neg O$, where O is the board position of the obstacle.

<u>*Number Constraints:*</u>

For clarity of naming in the rest of this section, with a given number tile 'A', the surrounding 4 tiles are B, C, D, and E. Number tiles have no effect on tiles diagonally adjacent to themselves– they only affect the tiles immediately above, below, to the right, and to the left of themselves.

<u>*0 Constraints:*</u>

For the number 0, no lights are to be placed next to the tile 'A'. This takes the form of [¬B] [¬C] [¬D] [¬E], all in separate clauses.

1 Constraints:

For a ‘1’ tile, there needs to be exactly one light in the surrounding squares. If in our example, B were chosen to be lit, this would generate the clauses B ⇒ ¬X, where X is a placeholder for any of the other available tiles surrounding A. This says that at most one neighbor tile has a light. We also need a case for “at least one tile needs a light,” so we include [B ∨ C ∨ D ∨ E]. (If a given tile were to be blocked, say C, this would instead be [B ∨ D ∨ E]). We don’t include blocked tiles.

2 Constraints:

For a ‘2’ tile, we need to specify that at least two of the surrounding tiles are true, and at least two of the surrounding tiles are false. For our example tile A with no obstacles near it, one example would look like B ∧ C ⇒ ¬D ∧ ¬E, which is equivalent to (¬B ∨ ¬C ∨ ¬D) ∧ (¬B ∨ ¬C ∨ ¬E). We would also need clauses of the form (B ∨ C ∨ D) ∧ (B ∨ C ∨ E), to ensure at least two of our tiles are true. We need to have both “at least two are true” and “at least two are false” in order to ensure that exactly two of the neighbor tiles are true. For each of these clauses, we actually need all the permutations of different tiles to be true. They follow the same general form as the one from the example, however, so we will not list them all here. If there is an obstacle next to a ‘2’ tile, the case becomes the same as a ‘1’ tile, except we concern ourselves with a “¬tile” instead of a light tile.

3 Constraints:

For a ‘3’ tile, we specify that at least 3 adjacent tiles are true, and at least one tile is false. To do so, we can use the logic from a ‘1’ tile, but with “nots” on all of the values. For a ‘3’ tile with no surrounding obstacles, we would use ¬B ⇒ X, where X loops through all of the other surrounding tiles. We would then also do ¬C ⇒ X, and so on. When there is one obstacle next to a ‘3’ tile, we simply set all available surrounding tiles to true.

4 Constraints:

For a number ‘4’, all surrounding tiles must contain a light. If this is not possible, the puzzle is unsolvable. For our example tile A, the constraints would look like [B] [C] [D] [E], which is the opposite of our 0 constraint clause.

Implementation:

During our implementation of this project, we ran into a few unexpected problems. Because we are working with a SAT solver, it was hard to know whether our constraints were valid until we had all of them coded. To deal with this issue for error-checking purposes, we had many print statements throughout the code to let us know how the constraints were being added or if any invalid constraints were being created. Another issue that we encountered while coding this assignment was the condition where a number constraint tile had one or more obstacles as a direct neighbor. In this case, we had to write special cases to check all of the neighbors of the tile and have different clauses depending on how many of them were obstacles. For example, a ‘2’ tile with an obstacle adjacent to it has clauses similar to a ‘1’ tile, where we move a single “not” tile into one of the neighboring positions instead of a light tile. For our “all tiles lit” case, we had to make sure that we included the current square that we were looking at in our set of visible tiles. We wrote a few functions to create booleans to determine if we were on an edge of the board, which we treated as obstacles in our adj_count() function.