Will Drake

1. Introduction

The game I chose to model was a tile slider puzzle. As a kid, I had my own small slider puzzle (Figure 1). I was thus motivated to create a game similar to this.

Figure 1:

Since I am very familiar with this puzzle, I wanted to model it and find the minimal solution for a specific randomized starting board. The game works as follows: There is a blank space that allows the player to move tiles. The goal is to take the randomized board and, by sliding numbered tile pieces around, get them in order from left to right, with the blank being in the bottom right.

2. Code, Formalization, and Analysis

For my tile game, I did a 3×3 to save monotonous writing of more tiles and a large number of moves or boards that would take a while to compile. However, since the fundamental moves for the tile game are the same, this code can be easily adapted to 4×4, 5×5 or beyond by simply changing the board and adding more tiles. For the code in Alloy, I started by using the tic tac toe model we did in class and adapted the code to work for my tile game. Similarly to tic tac toe, this tile game would be a 3×3 using row and columns; however, I also needed to map each row and col to a single tile. Thus, giving me my `sig Board` adapted from tic tac toe to

$$\text{Row} \; \rightarrow \; \text{Col} \; \rightarrow \; \text{one Tile}.$$

With my board ready, I created two predicates, one with the randomized `StartingBoard` and one with the `EndingBoard`; these would serve to work with my movement of the game to have a start and end. The movement of the game was the most challenging part and required the most thought. When you are playing the game in real life, you think of the movement as sliding a piece into an open space. To formalize this in Alloy, I needed to decide how movement could properly work for an Alloy implementation. Instead of thinking of it as sliding pieces, I thought of the blank space and a tile switching row and column positions. This can happen for any piece that is to the left, right, up, or down of the blank space.

I started by creating a `Neighbors` predicate, allowing me to identify the pieces that are on the left, right, up or down. For left and right pieces, the row would stay the same but the column would either be plus one or minus one, giving me the code.

$$(\, i_1 = i_2 \;\wedge\; (\, j_2 = \text{add}[\, j_1, 1\,] \;\vee\; j_2 = \text{sub}[\, j_1, 1\,]\,)).$$

For up and down the column would stay the same but the row would either be plus one or minus one, thus

$$(\, j_1 = j_2 \;\wedge\; (\, i_2 = \text{add}[\, i_1, 1\,] \;\vee\; i_2 = \text{sub}[\, i_1, 1\,]\,)).$$

Combining these with $\vee$ gives me all the neighbors possible. Next, I moved on to the `LegalMove` predicate. Having some `tile1` and some `tile2` with their respective row and col, I would have `tile1` be equal to the `Blank` and `tile2` be a neighbor of `tile1` for `board1`. Then I would set the `Blank` equal to the neighbor for `board2` and vice versa:

$$\text{some } i_1 : \text{Row},\ i_2 : \text{Row},\ j_1 : \text{Col},\ j_2 : \text{Col} \;\Big|\; b_1.\text{at}[i_1, j_1] = \text{Blank} \ \wedge\ \text{Neighbors}[\,i_1, j_1, i_2, j_2\,]$$
$$\wedge\ b_2.\text{at}[i_1, j_1] = b_1.\text{at}[i_2, j_2] \ \wedge\ b_2.\text{at}[i_2, j_2] = \text{Blank}.$$

Although this movement works perfectly, I need to ensure that no other tiles would change positions besides the blank and its neighbor, so I enforced that for all rows and columns, if they were not these two tiles, their position in `board1` and `board2` would be the same. So I added this onto `LegalMove`:

$$\text{all } r : \text{Row},\ c : \text{Col} \mid \neg\Big((r \to c) \in (\,i_1 \to j_1) \cup (i_2 \to j_2)\Big)$$
$$\implies b_2.\text{at}[r, c] = b_1.\text{at}[r, c].$$

With all this working I took the `Game` predicate with my `StartingBoard`, then applied `LegalMove` one board after another, ending with my `EndingBoard`. Through checking the game with different number of Boards, I discovered the `MinimalSolution` for my specific randomized tile slider was 23 moves.

3. Further Work

The next step with this game would be to look at different variations. The slider puzzle is always a square with the same amount of rows and columns, but what about looking at sliders with different amount of rows and columns? Are they still solvable in every state? If not, how many states are they solvable in? Are they solvable at all?