# **2048 Movement Mechanism**

# 20 (Slightly) Different 2048 VersionsProject Summary

The idea of the classical game of 2048 is that you move and merge tiles with equivalent values to generate tiles with larger values. We do not consider the merging functions, so we do not consider the numbers appearing on each tiles either. We are interested in only the movement mechanics of the tiles. The grid on which you can move the tile is a 4 \* 4 grid.

We want to start from a grid with only one randomly generated tile on it, and let the SAT solver determine in which of the four directions it can move and chooses a movable direction to implement the movement. We use a parameter `timeStep` to indicate our steps of movements. Since the grid starts with 1 tile at `timeStep` is `t\_0`, we expect the grid to be filled after 15 valid movements.

# Propositions

Each proposition will have a timestep property that represents the current move. There can only be 16 possible timesteps since the grid is 4x4.

# Location (loc, timeStep): A tile exists at location loc at timestep t

* MoveUp(timeStep): Checks if any tile can move up at the given timestep.
* MoveDown(timeStep): Checks if any tile in the grid can move down at the given timestep.
* MoveLeft(timeStep): Checks if any tile in the grid can move left at the given timestep.
* MoveRight(timeStep): Checks if any tile in the grid can move right at the given timestep.
* AbleToMove (loc, orientation, timeStep): Checks if the tile at location loc can move in the direction of the given orientation at the given timestep.
* Random (loc, timeStep): After a move, a random new tile is generated at a location loc. There are 16 minus timestep possible locations to place a tile.

# Constraints

* Location(loc, timeStep) Location(loc above, timeStep) AbleToMove(loc, up, timeStep): If we have a tile at a certain location and the location above that tile is empty, we are able to move up
* Location(loc, timeStep) Location(loc down, timeStep) AbleToMove(loc, down, timeStep)
* Location(loc, timeStep) Location(loc left, timeStep) AbleToMove(loc, left, timeStep)
* Location(loc, timeStep) Location(loc right, timeStep) AbleToMove(loc, right, timeStep)
* If tile on the first row AbleToMove(loc, up, timeStep): We are unable to move up at the first row
* If tile on the first col AbleToMove(loc, right, timeStep)
* If tile on the last row AbleToMove(loc, down, timeStep)
* If tile on the last col AbleToMove(loc, right, timeStep)
* Location(loc, timestep) AbleToMove(loc, up, timeStep) AbleToMove(loc, up, timeStep) AbleToMove(loc, up, timeStep) AbleToMove(loc, up, timeStep): If we do not have a tile at a location, it means that we are not able to move that non-existing tile along any directions
* Location(loc, timeStep) AbleToMove(loc, up, timeStep) MoveUp(timeStep) Location(farthest loc above, timestep + 1): If we have a tile at a certain location, and we are able to move up at the location, and we have actually moved up, we should have a tile at a new location upward (farthest as possible) at a new timestep.
* Location(loc, timeStep) AbleToMove(loc, down, timeStep) MoveDown(timeStep) Location(farthest loc below, timestep + 1)
* Location(loc, timeStep) AbleToMove(loc, left, timeStep) MoveLeft(timeStep) Location(farthest loc left, timestep + 1)
* Location(loc, timeStep) AbleToMove(loc, right, timeStep) MoveRight(timeStep) Location(farthest loc right, timestep + 1)
* Location(loc, timeStep) Random(loc, timestep + 1): If we already have a tile at a certain location, we are unable to generate a random new tile at that location.
* Location(random\_loc, timeStep) Random(random\_loc, timestep + 1): Once a random\_loc is picked, if we do not have a tile existing at random\_loc, we are able to generate a random tile at that location.
* A tile cannot move out of bounds (handled by our python code)

# Model Exploration

## Refining Our Propositions and Constraints

In our previous proposition, we used propositions U and L to declare the direction we are moving. However, we noticed that these propositions may be too hard to be used in the constraints and make things more complicated. Therefore, in our new propositions, we removed U and L. Instead, we used the parameter orientation to declare the direction.

In our previous proposition, to determine if two tiles can merge, we must check if, in the given direction, the adjacent next tile has the same value as the original one or the tile that has the same value on the row/column (depending on the direction) it is on, and the spaces between them are empty. This expression over here already seems too long and complicated. Therefore, we try to simplify the proposition by splitting it down. We introduce a new proposition called row\_or\_column\_can\_move (orientation) and we know that this is a result of some manipulations of our propositions. We then have able\_to\_move (t1, orientation) able\_to\_move (t2, orientation) able\_to\_move (t3, orientation) able\_to\_move (t4, orientation) row\_or\_column\_can\_move (orientation), which gives us the same result as we had before but using simpler combinations of propositions.

## Figuring Out the Data Type of the Tiles

Before we tried to create our tiles using types, i.e. we have type “2”, type “4”, … type “2024” tiles. However, it came to us that we need to deal with at most 16 tiles on the grid/board, therefore we could have a name for each of the 16 tiles and then try to model their moving mechanics. We thought at first we could use a dictionary of 16 entries to store the name of each tile as keys and then the values they bear as items. However, it came to us that it is difficult to create a new key and assign value to that key, and then update the value of the corresponding tile when a merge happens. Therefore, we have tried to figure out a new way to model the tiles. We are still progressing.

A computer screen with text

Description automatically generated

## Differentiating Each Tile from Another

Previously, we were using a dictionary to model our grid with the coordinates as keys and their numerical values as values. This turned out to be problematic as many of the constraints and propositions require us to check the coordinates. It would be hard to check the coordinates, so instead of them directly being the keys, we are creating tiles as objects, which would then be inserted as keys into the dictionary.

## Never Actually Updating the Grid (We used a recursive moving function here, later replaced by a recursive distance function)

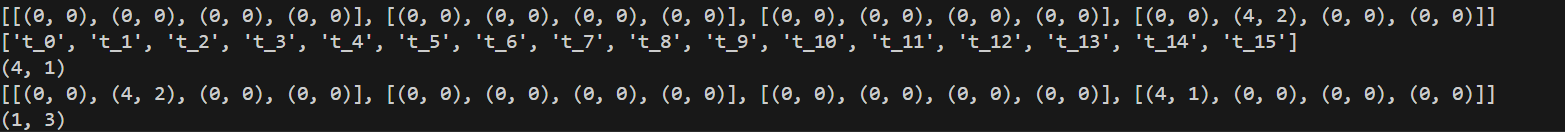
we wrote a function that make movements recursively in UP direction to test our data structure and our recursive movement function. Since our grid is initialized with one value on it, after making one movement, provided that the initialized block is in row 1, we should expect grid to be modified so that it has two valid locations in it. However, we get only 1 valid location, which is the initialized location being moved UP. We figure that the second "if" statement is never entered in Move(orientation). Through some testing, we suspect that that GridTemp might always change with GRID, so GRID == GridTemp is always true, thus never entering the if statement. We fix it by making a deep copy correctly.

A computer screen shot of a program code

Description automatically generated

## Tile Being Moved but Coordinates Not Updated

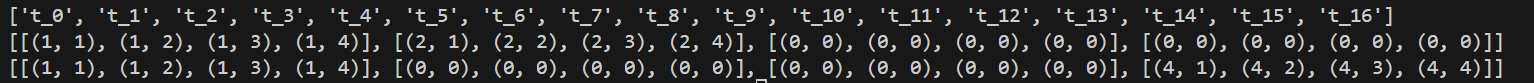
When we move a tile, the moved tuple should be updated corresponding to the new location, instead of simply moving the tuple's position in the list (i.e. when we move (4,2) to the first row, it should become (1, 2)).



We fix this bug by modifying the recursive movement function.

## Two Tiles Stacked Together Unable to Move Down/Right at Same TimeStep (Version I)

We found that for downward movement, this situation happens because the recursion starts from the top to bottom, so even if at the deep bottom there are spaces to move, the program would not know until it goes to the bottom at the end, resulting in the weird result that only one row is moved.



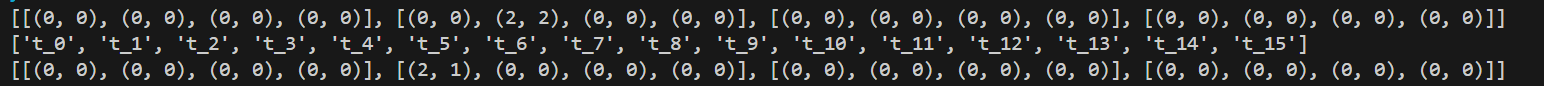
We add a another for loop in front of the nested for loops so that the recursive downMove function is called three times, once for the first three rows. (Same strategy used to fix rightMove) This would solve our problem by now but we encounter the problem of the same kind with the SAT solver later again.

A screen shot of a computer code

Description automatically generated

## Random Tile Not Generating Properly

In this case we are testing, when (2, 2) moves to (2, 1), a new tile is not generated.



We found that the list of random locations is not generated properly. We mess up with the order and number of for loops. Fix it by rewriting the loops.

## Contradictions With Our Constraints (New Constraints Added but Old Ones Not Nullified)

When a tile has moved, the SAT solver is aware of the new location of the tile being moved, but it does not cancel its previous location. For example, when (2, 1) moves down one block to (3, 1), the SAT solver says (3, 1) is occupied but it also says that (2, 1) is still occupied, which is wrong.

A screen shot of a computer

Description automatically generated

We first thought about trying to delete a proposition or a constraint previously added to the theory. However, we talked to our TA Connor and found that it was not suggested to do that. Then we tried to fix this with our timeSteps. We found that we added the constraint Location(2, 1, t1) in the beginning, and now after the movement, a new constraint Location(3, 1, t1) is added. We have not differentiate the constraints with different timeSteps. We fix it by adding Location(2, 1, t0) in the beginning, adding Location(3, 1, t1) after the movement, and also ~Location(2, 1, t1). In this way, the SAT solver knows that at time t1 (2, 1) is now empty and (3, 1) is taken.

## Two Tiles Stacked Together Unable to Move Down/Right at Same TimeStep (Version II with SAT Solver)

We encounter the same problem we have seen before. We have two tiles stacked together, in this case we have (1, 4), (2, 4). However, when we try to move down, only the lower tile in the two tiles stack is actually moving. In this case, (2, 4) moves to (3, 4) but (1, 4) stays in the same place.

A black screen with white text

Description automatically generated

We found that the reason is that since our DistanceDown function checks recursively all our tiles from top down and left right, the SAT solver does not have (1, 4) ableToMove so it does not move. We fix the problem by looping from the bottom to the top when we check for the distance moving down, and we fix a similar problem involving moving right by looping from the right to the left.

A screen shot of a computer screen

Description automatically generated

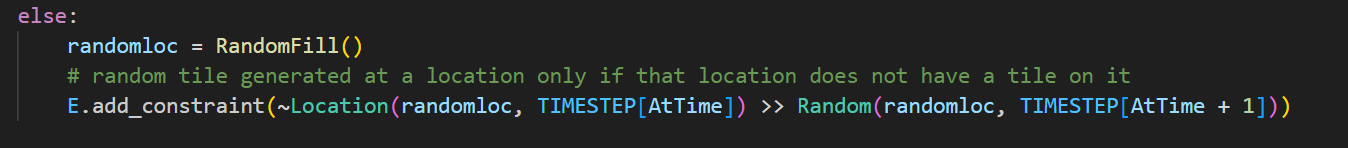
Random Tile is Generated But the SAT Solver and the Grid Do Not Match

In our testing case, after moveUp is true, we have (2, 1) moved to (1, 1), and the SAT solver tells us that a new tile (4, 3) is generated. However, our grid is not updated.

A black screen with white dots

Description automatically generated

We fix this problem by calling RandomFill() in our code and then add the constraint implying Random().



A computer screen with colorful text

Description automatically generated

# /Users/andrewzhang/Desktop/prooooof/Screenshot 2024-12-08 at 19.59.54.pngScreenshot 2024-12-08 at 19.59.54Jape Proofs

## Restrictions on Moving Directions

If tiles are not oriented ULD, then it must be R.

We make following assumptions:

* Not oriented ULD:  
   ¬U, ¬L, ¬D
* Must move towards one of the directions:  
   U ∨ L ∨ D ∨ R

We want to prove that R holds:

This is our Jape proof

## Cannot Generate Random Tiles When Board is Full

If all of the locations are filled up, then we cannot generate a random tile.

We can set up our premises

- If for any line x, for any column number y such that (x,y) holds:

x.y.(PL(x, y))

- If all PL(x,y) (x[1,4], y[1,4]) holds we cannot generate a random tile

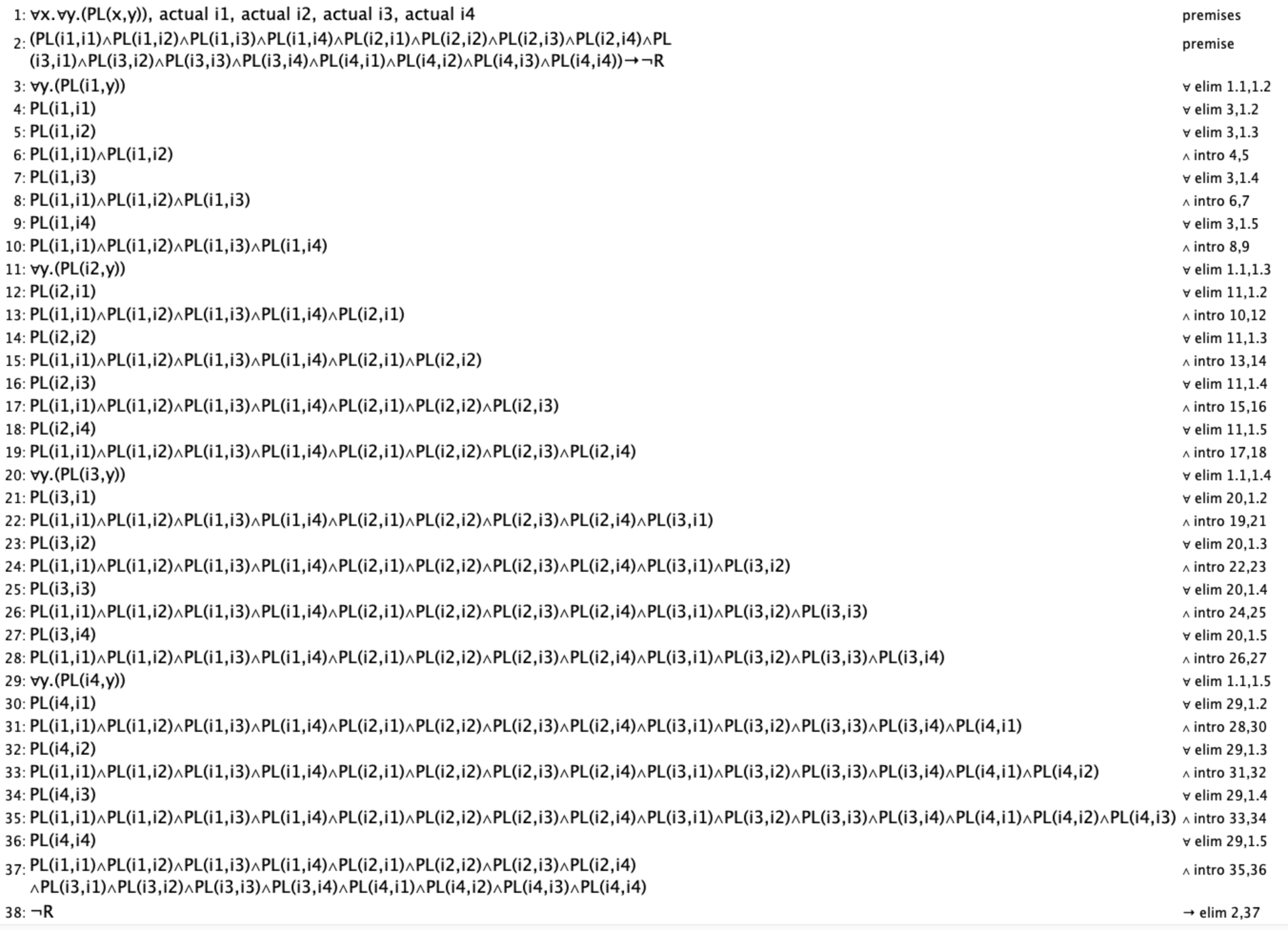
(PL(i1, i1) ∧ PL(i1, i2) ∧ PL(i1, i3) ∧ PL(i1, i4) ∧ PL(i2, i1) ∧ PL(i2, i2) ∧ PL(i2, i3) ∧ PL(i2, i4) ∧ PL(i3, i1) ∧ PL(i3, i2) ∧ PL(i3, i3) ∧ PL(i3, i4) ∧ PL(i4, i1) ∧ PL(i4, i2) ∧ PL(i4, i3) ∧ PL(i4, i4)) → ¬R

-Now we can set up our initial configurations:

actual i1, actual i2, actual i3, actual i4 (these variables represents row and column numbers)

We want to prove that ¬R holds.

This is our proof:



## Application of Location Reachability

If we move right, then for any line, if there exist a tile at the very left block (first column), and all other blocks on that line are empty, then the tile will move to the very right block (forth column)

We can set up our premises

- If we move right, then for any line x, if the first column is filled, and the second column is empty, then the tile in the first column will move to the second column, leaving the first one empty.

PMR→∀x.((PL(x,i1)∧¬PL(x,i2))→PL(x,i2)∧¬PL(x,i1))

- Similarly, If we move right, then for any line x, if the second column is filled, and the third column is empty, then the tile in the second column will move to the third column, leaving the second one empty.

PMR→∀x.((PL(x,i2)∧¬PL(x,i3))→PL(x,i3)∧¬PL(x,i2))

- Similarly, If we move right, then for any line x, if the second column is filled, and the third column is empty, then the tile in the second column will move to the third column, leaving the second one empty.

PMR→∀x.((PL(x,i2)∧¬PL(x,i3))→PL(x,i3)∧¬PL(x,i2))

- Likewise, If we move right, then for any line x, if the third column is filled, and the fourth column is empty, then the tile in the second column will move to the fourth column, leaving the third one empty.

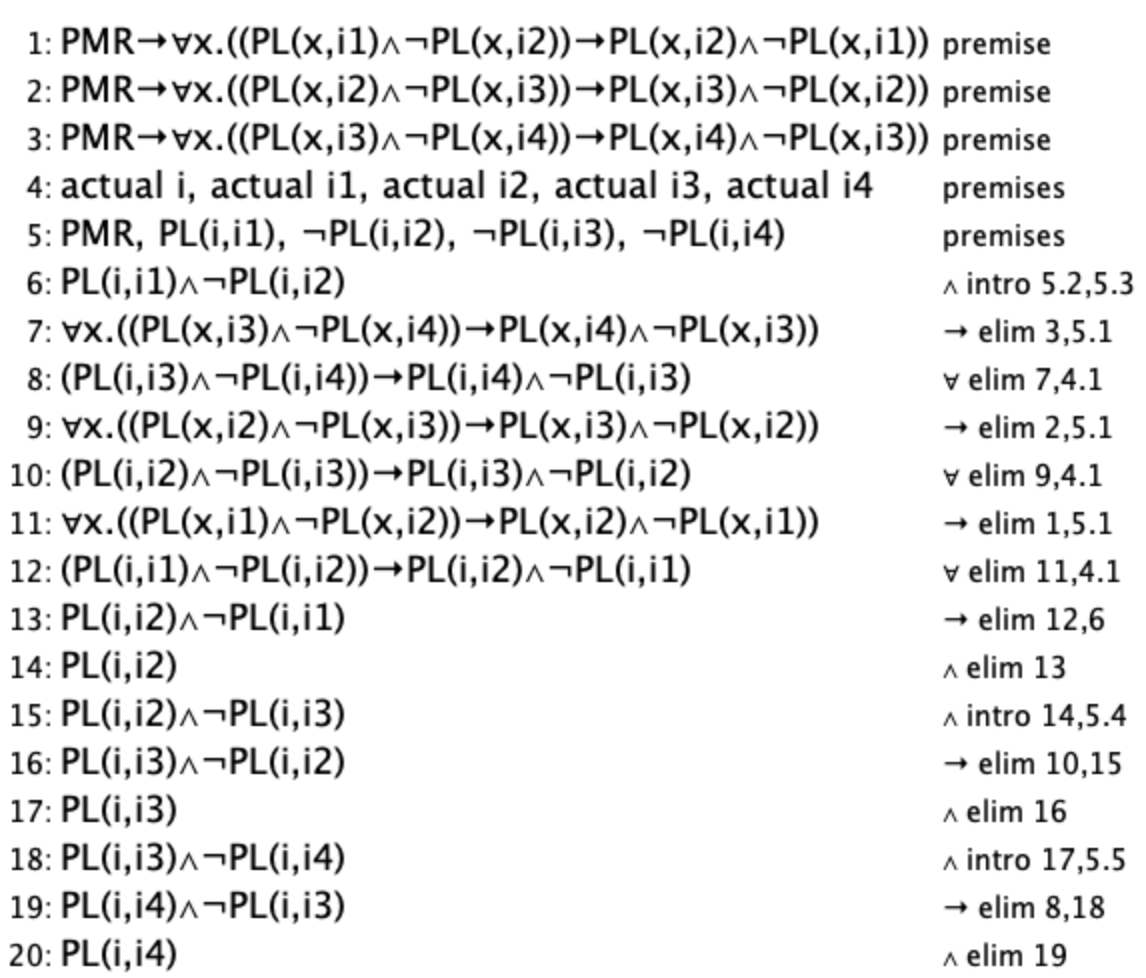
PMR→∀x.((PL(x,i3)∧¬PL(x,i4))→PL(x,i4)∧¬PL(x,i3))

- Then we can set up our initial configuration:

actual i1, actual i2, actual i3, actual i4 (these variables represents row and column numbers), PMR, PL(i,i1),¬PL(i,i2), ¬PL(i,i3), ¬PL(i,i4).

We want to prove that PL(i,i4) holds.

This is our proof:



# First-Order Extension

Predicates that we already have:

# Location (x, L, t): x is at location L at timestep t.

* MoveUp(t): Any of the tiles on the board can move up at timestep t.
* MoveDown(t): Any of the tiles on the board can move down at timestep t.
* MoveLeft(t): Any of the tiles on the board can move left at timestep t.
* MoveRight(t): Any of the tiles on the board can move right at timestep t.
* AbleToMove (x, L, t): Checks if some x on location L can move in any direction at a given timestep.
* random (loc, timeStep): A new tile is at a location loc.

Predicates that will help us talk about tiles and predicates that will help us with the formulae:

* Tile(x): x is a tile
* REdge(x): x is on the rightmost boundary.
* LEdge(x): x is on the leftmost boundary.
* UEdge(x): x is on the uppermost boundary.
* DEdge(x): x is on the lowermost boundary.
* RMove (x): x can move to the right.
* LMove (x): x can move to the left.
* UMove (x): x can move up.
* DMove (x): x can move down.

Constraints represented in First-order logic:

## A Tile cannot move out of bounds

Any tile that is on an edge can’t move further out of bounds. Best to break this down into multiple formulae: (1) A tile on the right edge can’t move in the right direction, (2) A tile on the left edge can’t move in the left direction, (3) A tile on the upper edge can’t move in the upper direction, (4) A tile on the low edge can’t move in the lower direction.

1. Tiles on the rightmost edge cannot move right: )
2. Tiles on the leftmost edge cannot move left: )
3. )
4. )

The final constraint would just be .

In contrast, this is what the Bauhaus implementation looked like to capture this constraint:

A computer screen shot of a program

Description automatically generated

## No duplicate tiles

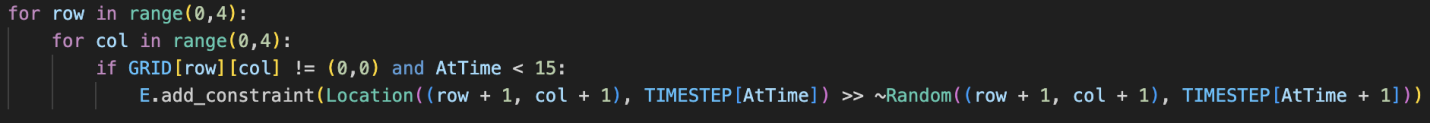
Any location can only have one tile at any time step.

## Random tiles must be unique

Each new random tile must be placed at an empty location. Note: A new random tile can only be generated if the timestep is less than 16. It would be better to break down this formula: {1} If there exists a tile at location L or the timestep is more than 15, then a random tile cannot be generated at the given location at timestep t+1, {2} if no tile exists at location L and timestep is less than 15 than a random tile can be generated at the given location.

The final constraint would just be

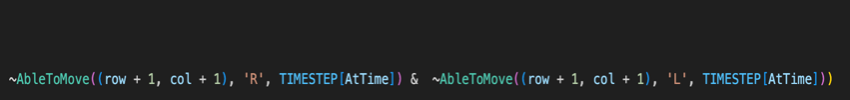
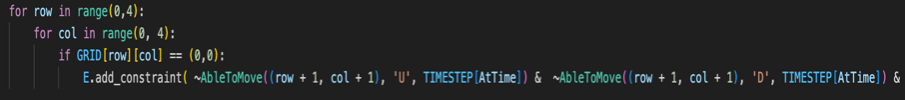
In contrast, this is what the Bauhaus implementation looked like to capture this constraint:



## Only tiles can move

An object’s location can only change if it is a tile. It would be better to break down this formula: {1}: if there exists a tile at the location, that tile can move, {2} If a location is empty, no movement is possible at that location.

The final constraint would just be

In contrast, this is what the Bauhaus implementation looked like to capture this constraint:

## Empty tiles imply movement

Tiles can continue moving until there are no more empty tiles left.