# Slide Puzzle Logic Modeling Project

# Group 9, CISC 204 Project, Fall 2023

# Project Summary

Armin Heirani (20350356), Ikra Mortoza (20350748), Serena Emond (20164315)

Our project models a 3x3 slide puzzle - a puzzle in which there is a 3x3 board with 8 numbered tiles placed randomly in eight of the nine positions. The player must slide the tiles around on the board to reach an end configuration where the tiles are sorted from top to bottom in ascending order, from 1 to 8, with the blank space in the bottom right corner.



In this project, a model corresponds to a 3x3 board with numbered tiles placed in 8 of the 9 positions. The program aims to evaluate whether a given initial board can be solved in a specific number of moves/time steps, and if so, output the moves it would use to solve the puzzle. The puzzle is modelled in a propositional logic setting by having propositions corresponding to where each tile is at a given time, as well as propositions corresponding to what moves need to happen at each time step, in order to satisfy a proposition that defines the winning layout of the board. Our model will output a solution consisting of assignments to the swapped propositions, which tell us which tiles/positions were swapped at each time of the puzzle, and the resulting assignments to the propositions about where each tile is located at each time step, to satisfy the goal constraint where all the numbered tiles are in their correct positions.

# Propositions

* Apijt: True when tile p is in the position (i,j) at time t.
  + This proposition checks whether a tile is in a position at time t.
* C: True when the number of swaps that have occurred is less than the specified number of swaps given to get from the initial state to the goal state.
  + This takes in the number of swaps that have occurred at that time and the number of minimum swaps required to get from the initial state to the goal state (called max\_swaps because that is the maximum number of swaps the model should have to get the best possible moves to solve the board).
  + C(s, m), where ‘s’ is the number of swaps that have occurred at that time and ‘m’ is the number of minimum swaps required to get from the initial state to the goal state.
* Sijpq: True if the tile in position (i,j) at time t and the tile in position (p,q) at time t should swap at time t+1.
  + This keeps track of which tiles have swapped at a specific time of the puzzle.
  + **swapped**((i, j), (p, q), **timer\_add**(t), **board\_updater**((i, j), (p, q), t, E), **clock\_updater**(E, t + 1))
  + The swapped proposition in our code contains the following:
    - The two positions where tiles were swapped.
    - The new time value, which is the value returned by the timer\_add() function.
    - The model E which is returned by the board\_updater() function with the newly added values for the assigned proposition at time t+1 for all positions that are not the ones just swapped.
    - And the model E again with the added value for the clock proposition at time t+1 which is returned by the clock\_updater() function.
* W: True when the board is in its goal state.

# Constraints

* The board can only be in its goal state if and only If W is true, where all of the tiles are assigned to their correct position on the board such that:
  + W >> (A1,00,m & A2,01,m &…& A’blank’,22,m & C), where m is the max number of swaps required to win the board.
* This implication works both ways, so (A1,00,m & A2,01,m &…& A’blank’,22,m & C >> W)
* We also added constraints which assign the tiles to positions on the board at the time of the board. These constraints ensured that if a tile x was in the position (i, j) at time t, then all other tiles were not in that position (i, j) at time t.
  + ∀t∀x∀p(A(t,x,p)→∀y(¬E(x,y)→¬A(t,y,p)))
    - Where t is a time, x and y are valid tiles, and p is a position (i,j) on the board.
    - E(y,x) is true when x==y, such that they are the same tile.
* At time t, two tiles have to be assigned to adjacent positions, where one tile must be the ‘blank’ tile, and the clock is true such that it has not reached it’s maximum number of swaps, then the tiles can swap at time t+1 where the two tiles are now assigned to each other's position at time t+1 and all other tiles are then assigned at time t+1 to the same positions it was assigned to at time t.
  + For example (Note: **Assigned** is our **Apijt** proposition, **swapped** is our **Sijpq** proposition, and **clock** is our **C** proposition):

**(Assigned**(x, (0, 0), swaptimer) & **Assigned**('blank', (0, 1), swaptimer) & **clock**(swaptimer, max\_swaps)) >> (**Assigned**('blank', (0, 0), swaptimer+1) & **Assigned**(x, (0, 1), swaptimer+1) & **swapped**((0, 0), (0, 1), **timer\_add**(swaptimer), **board\_updater**((0, 0), (0, 1), swaptimer, E), **clock\_updater**(E, swaptimer + 1)))

* + The x in the assigned proposition is any tile in the list of tiles, and this constraint is iterated over for all tiles, x. We also have this constraint for all of the pairs of possible positions where tiles can swap.
  + This constraint ensures that only the ‘blank’ tile is ever swapped with another tile. This is important because the rules of slide puzzle state that a tile can only move into a blank position, and this is the way we enforced this rule into this constraint.

# Model Exploration

We initially wanted to set up our model so that, given an initial board, the model should give us a list of moves that solves the board. The initial board we input would assign tiles to positions initially, and then the model would change the assignments with each move (using our Assigned propositions). This led us to then think about what we mean by “moves.” In a slide puzzle, tiles are moved around by sliding them into the blank space. This caused us to change how we looked at the problem: at first we were viewing “moves” as tiles swapping with each other to get to the right positions, however we simplified the setting by viewing it as the “blank” tile swapping with the neighbouring number tiles repeatedly (since a tile can only move into a blank space). So we initially had propositions Bij (tile swapping is blank), and Uij, Dij, Lij, Rij (tiles above, below, left, and right of the tile to be swapped are valid tiles on the board). So this new perspective on moves led us to reformat our constraints so that we set Bij as always true, and thus, we do not need to check whether the tile swapping with the initial tile is blank, since the initial tile will always be the ‘blank’ tile. Specifically, when assessing the above, below, left, and right propositions, we did not need to add the constraint that Bpq had to be true for the direction propositions (Uij , Dij , Lij , Rij) to be true. This saves us a lot of computing time because the model would not have to check each tile on the board to see which ones were neighbouring a blank and thus could move into the spot.

Eventually, however, we ended up simplifying our problem more, by further changing how we were modelling the swaps in logic. Initially, we had had a proposition T which confirmed if a position was a valid position on the board, as well as U, L, R, D which checked if the position above, left, right, or below the current position was a valid position on the board by checking the indexes (since we didn’t want our program to move tiles off the board). Our constraint then was that for a swap to be able to happen, B (blank) had to be true, T had to be true and U or L or R or D had to be true. We simplified this by first adding an external file to hold all the valid board positions called Board.py and a file to hold all our tiles (1-8 plus the “blank” tile) which ensured that the program would always be assigning a tile to one of the positions and not one that doesn’t exist. However, as we were encountering errors while running the file, we realized that our old propositions were not suited to the problem we were trying to solve and did not allow us to structure our constraints the way we wanted. This led us to changing our swap proposition, S, into 12 different propositions called swap\_posxposy (x and y were substituted with all the different valid adjacent position pairings – for example, swappos1pos2 exists, but swappos1pos5 does not). Now in the constraints, we ensured that swapping x and y could only happen if x or y was assigned to “blank”, and it would result in the assigned propositions of the two switching tiles. With this new change, we got rid of our T, U, L, R, and D propositions (we ended up changing the formatting of this to arrays we joined with an and function later).

After some feedback from our TA and peers, we ended up adding a time factor to our project to increase its complexity, which counts the number of swaps. The idea is that initially, the time = 0, and it increments by 1 after each move. The aim is to input a specific time (which we call max\_swaps) along with the initial board, and the program should be able to determine whether a solution is possible within that number of swaps and output the solution if so.

We started to run into problems when we realized that we could not assign a tile to a position and then unassign it as we were trying to do to make tiles swap. This is because a proposition cannot be both true and false in the model. This led us to add the time parameter to many of the propositions, including the assigned proposition, so that a tile can be at a position at time 1 and not at that position at time 2, which allows us to swap tiles and keep track of where the tiles are on the board at each swap.

Additionally, while creating the time variable, we ran into errors where the time was not updating at the times that we set it to update. We realized that this is because in the build\_theory() function, it only runs once and so it cannot keep updating to reflect the state of the board. To solve this problem, we created a few functions which were called at the moment that the values should have changed. The time updating function, the clock\_updater function (which updated the clock parameter for the new time), and the board\_updater function (which sets the new values of which tiles and positions are assigned together at the new time), were all called when a swapped parameter was true. This swapped parameter replaced our other swap parameter because swapped only returns true when two tiles have swapped at a given time. This allowed us to format our constraints in such a way that all the functions are called at the appropriate times and the model can tell us which tiles were swapped and in which order (time of the puzzle).

Given these changes, our model tracks which swaps were performed at what time, and where each tile is at each time. So then, a solution would output the positions of each tile at each time on the board by the assigned proposition, and the swaps that have occurred at each time of the puzzle until max\_swaps. For example, the model would output at time 0 which swaps are true, at time 1 which swaps are true, all the way to time max\_swaps, to satisfy the constraint of where all the tiles should be that implies a win.

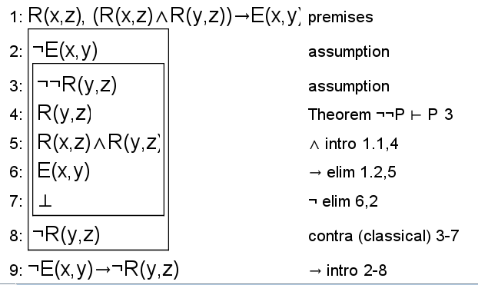
# Jape Proofs

**Proof 1:**

One constraint we ended up having to implement is that if a tile is assigned to a position at time t, then a different tile cannot be assigned to that position at time t. This is because of our constraint that only one tile can be assigned to one position at a time (i.e. if x and y are assigned to the same position at time t, then x and y must be the same tile).

So if x and y are valid tiles, and z is a valid position on the board, and we have premise R(x,z): tile x is at position z, and the constraint as our second premise, we should be able to conclude that if a tile y is not the same as tile x then y cannot be assigned to the position z.

R(x,z), (R(x,z)∧R(y,z))→E(x,y) ⊢ ¬E(x,y)→¬R(y,z)



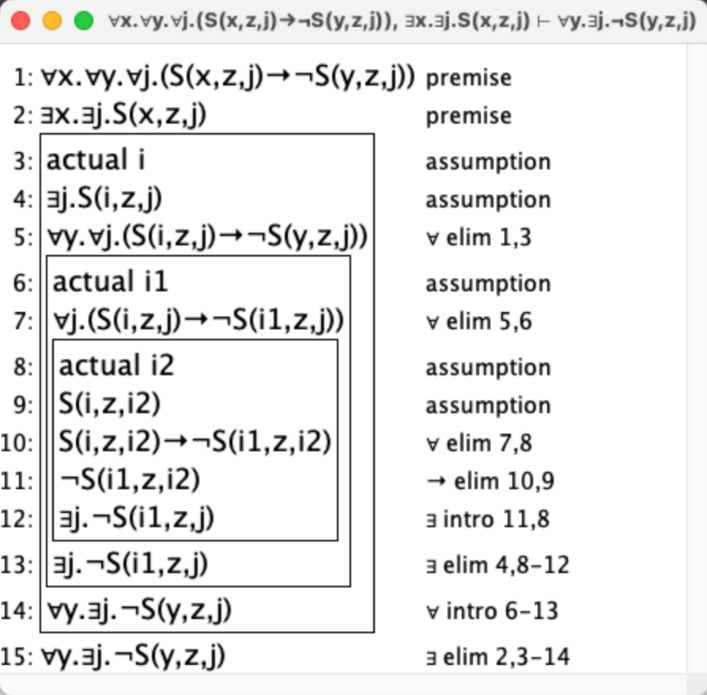
**Proof 2:**

For this proof, we wanted to show how only one swap is allowed per time of the puzzle, where a tile only swaps with the blank tile. For all tiles on the board (x), all other tiles on the board not including x (y), and all times of the puzzle (j), if a tile is swapped with a blank tile (z) at a given time, all other tiles must not be swapped with the blank tile at that time. This proof shows how only one swap is allowed per time of the puzzle, where if one tile swaps with the blank at time 1, then no other tiles will have swapped with the blank at that time.

We set the swapped proposition as S(x,z,j) and S(y,z,j), where x is a swapped tile, y is any tile that is not x, z is the blank tile, and j is the time of the puzzle.

**∀x.∀y.∀j.(S(x,z,j)→¬S(y,z,j)), ∃x.∃j.S(x,z,j) ⊢ ∀y.∃j.¬S(y,z,j)**

See the proof below:



**Proof 3:**

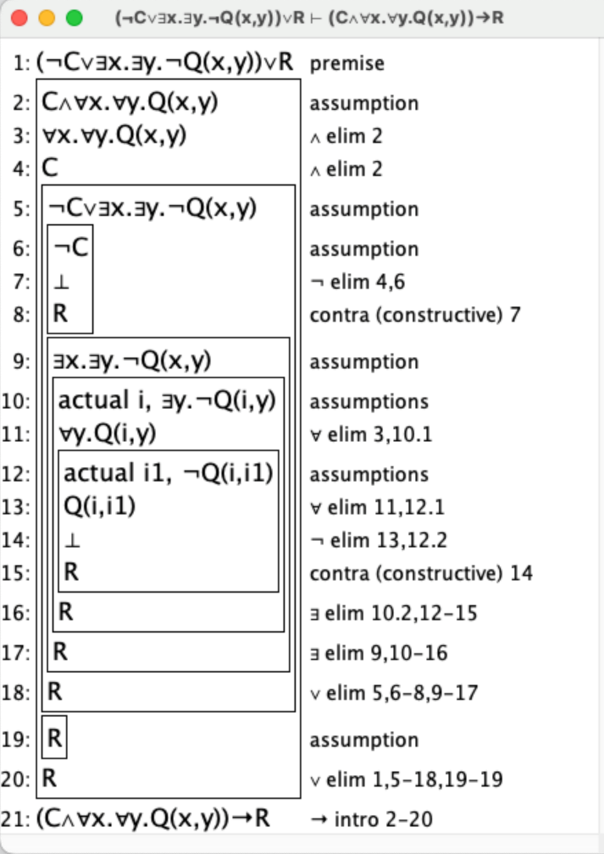
We wanted to prove that a board is won (W) when all tiles are in their correct positions and the number of swaps that have occurred in the model is equal to the minimum number of moves required to solve the board (clock). We prove this from the statement that the board is either won or the number of swaps that have occurred in the model is not equal to the minimum number of swaps required to solve the board (clock) or there exists a tile, x, and a position, y, which is not assigned to its correct position on the board.

We chose to do this proof because we wanted to show when the board is won and show what the conditions are that would make it so that the board is not won.

Because of the restrictions in jape, we had to make the winning condition be R, and the correct tile-position assignment be Q.

(¬C ∨ ∃x.∃y.¬Q(x,y)) ∨ R ⊢ (C ∧ ∀x.∀y.Q(x,y)) → R

Below is the solved jape proof:



We solved this proof by an implication introduction, a disjunction elimination, which solved the bottom of the proof, then two conjunction eliminations, another disjunction elimination, a negation elimination which introduces bottom and a contra constructive to solve the top box. Then, we did 2 existential eliminations, 2 universal eliminations, another negation elimination which introduces bottom, and then a contra constructive to finish off the proof.

# First-Order Extension

To reframe our propositions in a predicate logic setting:

* A(x, p, t): true when tile x is at position p at time t
* S(x, p, y, q): true if the tile x at position p and the tile y at position q at time t swapped at time t+1
* C(s, m): true when the number of swaps s that have happened is less than the target number of max swaps m inputted initially
* W(x\_1, x\_2, x\_3, x\_4, x\_5, x\_6, x\_7, x\_8, B): true when all tiles x\_i and B (blank spot) are in their correct positions.

For our model, the universe of discourse would be the set of all valid tiles (T\_i), the blank space (B), all valid board positions (P\_i), and the integers from 0 to the given maximum number of swaps (to represent the time steps).

One constraint we require is when it comes to swapping – swaps can only happen with the blank space and a tile adjacent to it. So we could define the S predicate:

S = { (B, P\_1, y, P\_2), (B, P\_1, y, P\_4), (B, P\_2, y, P\_3), (B, P\_2, y, P\_5), (B, P\_3, y, P\_6), (B, P\_4, y, P\_5), (B, P\_4, y, P\_7), (B, P\_5, y, P\_6), (B, P\_5, y, P\_8), (B, P\_6, y, P\_9), (B, P\_7, y, P\_8), (B, P\_8, y, P\_9), (x, P\_1, B, P\_2), (x, P\_1, B, P\_4), (x, P\_2, B, P\_3), (x, P\_2, B, P\_5), (x, P\_3, B, P\_6), (x, P\_4, B, P\_5), (x, P\_4, B, P\_7), (x, P\_5, B, P\_6), (x, P\_5, B, P\_8), (x, P\_6, B, P\_9), (x, P\_7, B, P\_8), (x, P\_8, B, P\_9)}

If we extended this by adding enough to substitute x and y with the tile numbers, it would be the list of all legal swaps.

Another constraint of ours we had to define using predicate logic already – the rule that if there is a tile at a position at time t, then no other tile can be assigned to that position at time t:

∀t∀x∀p(A(t,x,p)→∀y(¬E(x,y)→¬A(t,y,p)))

The predicates used here are A(t,x,p), which is true if tile x is assigned to position p at time t, and E(x,y), which is true if tiles x and y are the same tile.

Another constraint is that if at time t two tiles are assigned to adjacent positions, and the clock is true such that it has not reached the maximum number of swaps, then the tiles can swap. To model this in predicate logic, we can introduce a new predicate that tells us if two tiles are adjacent, N(x,y). Then:

∀t∀s∀m∃x∃y(N(x,y)∧C(s,m)) → S(x,p,y,q)

One way we would apply the predicate logic setting to a possible extension of our model is by modelling the number of inversions on a board. In the context of our game, an inversion refers to a pair of tiles that are in the reverse order of their goal or sorted order. For a sequence of distinct elements (e.g., numbers or tiles), an inversion occurs when two tiles a and b are in positions i and j within the sequence, and i<j, but a>b. This is helpful to determine whether an initial puzzle board is solvable or not – if the number of inversions on a 3x3 board is even, only then it is solvable.

To express the concept of an "odd number of inversions" in the context of the 8-puzzle problem using predicate logic, you can define predicates and then formulate the proposition. Let's break it down:

Let's denote the tiles on the board as *T*1​, *T*2​, …, *T*8​ and the blank tile as *B*.

**Predicates and Constraints:**

* Ti represents each tile in the puzzle.
* B represents the blank tile.
* Grid represents the 3x3 grid.
* Position(Ti) represents the position of tile Ti.
* Move(Ti) represents a move action of tile Ti.
* Adjacent(Ti, Tj) is a predicate indicating that tile Ti is adjacent to tile Tj.
* Value(Ti) represents the value of tile Ti.
* Inversion(Ti, Tj) is a predicate indicating that there is an inversion between tiles Ti and Tj.
* Even(x) and Odd(x) are predicates indicating that x is even or odd, respectively.
* Row(B) represents the row number of the blank tile B.
* Configuration(a) represents a specific configuration ‘a’ of the puzzle.
* Goal represents the goal state of the puzzle.
* L(Ti, Tj): Predicate indicating that tile Ti is to the left of tile Tj.
* I(Ti, Tj): Predicate indicating that there is an inversion between tiles Ti and Tj.
* For each pair of tiles Ti and Tj such that i < j, if (Ti, Tj) and Ti and Tj are in reverse order (inversion), then I(Ti, Tj).

**Tile Positioning:**

**Tile Movement:**

**Inversion Order:**

Formally, we can express this as:

*We assume that*

**Unique Tiles:**

**Goal State:**



# Next Steps for Our Project

At the time of submission, our project is still experiencing some bugs. For one thing, something we still need to implement is getting the tiles to not be assigned to the same position at a given time. We attempted to do this by adding this to our code, where t1 and t2 are the tiles that switch, and pos1 and pos2 are the positions of those tiles:

for tile0 in TILES:

if t1!= tile0:

E.add\_constraint(~**Assigned**(tile0, pos1, swaptimer+1))

if t2 != tile0:

E.add\_constraint(~**Assigned**(tile0, pos2, swaptimer+1))

Also we were unable to implement the reverse order of the positions where the blank tile is also in the position where the tile was.

In our project, our run\_w\_clock.py file is where we attempted to solve these problems and more, which did lead to more bugs, and we will continue to investigate until the project is in its final form.

Another thing we will do is to make the model output/print the board for each time of the puzzle. This would help readers/users to better understand what swaps occur and where, and the number of swaps that occur.

Once the project is in its final stage, there are also some interesting expansions we could look at. One is that we could apply it to boards that are of any size, rather than just 3x3. The basic structure and algorithm would be the same, just applied recursively for the extra rows and columns. We could also look into having our program take in an input board from the user, and check if the board is solvable first (an n-by-n board is unsolvable if the number of inversions initially is even if n is odd, and if the number of inversions is odd if n is even). Another possible expansion is rather than outputting a solution to the board under a specified number of moves, it could determine the solution with the minimum number of moves required.