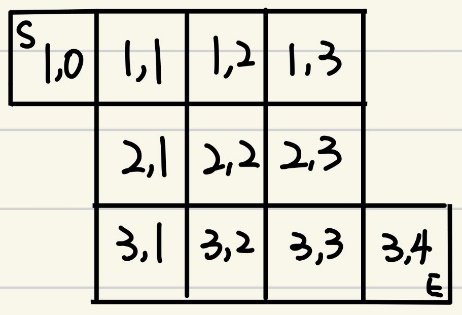
# Project Summary

*This is a game about connecting pipes in a 3x3 grid, where the start is in the top left corner and the end is in the bottom right corner. The goal of this game is to connect the starting pipe, the ending pipe, and every pipe in between them (there are three types) by rotating the pipes on the grid. We made a simple diagram to get a better idea of ​​what the entire grid looks like. We will simulate how pipes are placed, and how they are connected in the end.*

# *A screenshot of rico's chiliwork(pipe puzzle)*Propositions

*TODO*

1. Check if we need more propositions
2. Refinement of Variable

Draft:

* pipe\_type (): In addition to the start and end points, which have only one opening, there are three other styles, as shown above, Straight line, corner type, and T shape. If we recognize that it's a two-opening type with WE or NS in one location, we randomly choose from option 1. If we recognize that it's a three-opening type, we randomly choose from option 2. Otherwise, we choose from option 3.
* Location (pipe\_type, loc): pipe is at location. The location is where the board that things exist.
* Orientation (): We have four directions on each of the grids, which are east, west north and south, represented by “NSEW”.
* Neighbor (loc): Indicates neighboring grids, e.g. [11, 12]
* Neighbor\_updown(Neighbor, pipe\_type): For up and down adjacent grids, there exist pipes that can be connected (with corresponding opening directions)
* Neighbor\_leftright(Neighbor, pipe\_type): For left and right adjacent grids, there exist pipes that can be connected (with corresponding opening directions)
* Grid\_position\_connected(Neighbor, PIPE\_TYPE): one position has south and the other has north (two cells connected)
* Solver runs one configuration

Solver (loc, PIPE\_TYPE, Orientation): A particular pipe is oriented in a particular way. When you run the solver, it’s going to say your configuration and the directions.

# Constraints

*TODO*

1. Check if we need more constraints.
2. Complete the corresponding jape proof.

Draft:

* In a grid, only one pipe with exactly one config can exist.

¬ location(pipe\_type1, loc) \/ ¬ location(pipe\_type2, loc)

* Two faces are connected if they are each adjacent and have lines facing each other.

Neighbor ()Connect ()

* We need at least one solution to exist, and we can stop when checking for a feasible route.
* Win condition: A transitive connection from the starting pipe to the ending pipe

Connected(start)Connected1Connected2 … Connected(end)

* There is only one possible pipe type for the start(E) and one for the end point(W).
* The pipe type of the start and end points can only be on grids “01” and “34” respectively.
* Pipe\_connect: after we randomize the setup, check to see if each grid is connected.
* There is a limit to the pipe next to the starting and ending pipe. They can't be in a situation where can't connect.

# Model Exploration

For one of our constraint pipe\_type, we write a nested for loop to find every possible figure a pipe can have (like [‘N’,’ W’], but the elements in there should not be the same). Nested for loop for i, for j, for k when running giving back something like [‘N’,’ W’,’ W’] which is not the expectation. Then we realized something was wrong in the j and k loops since they are repeated ones. After correcting the staring value in loops from (0,i+1,i+2) to (0,i+1,j+1). The previous nested loop goes over the second part of the array twice which causes j and k loops to form the same element.

We have this constraint connected which generates with for loop what kind of a pair of pipe\_type can be connected like (pipe with opening to E and opening EW). We found it returned [[‘E’] [‘W’]] which means that the start and end are connected which is not possible. Since that is the only case that is possible in the whole generated array, we just delete that from the array

# Jape Proof Ideas

1. When there are left and right connected grids, it may not be necessary to look at the up and down of the left grid to see whether they are connected or not.
2. No need to consider optimal solutions.

# Requested Feedback

1. We have no idea how to start on Jape, can you tell us how it should look on Jape code?
2. Do the propositions and constraints make sense? (like can you picture how the game goes and win)

# First-Order Extension

*Describe how you might extend your model to a predicate logic setting, including how both the propositions and constraints would be updated.* ***There is no need to implement this extension!***

# Useful Notation

*Feel free to copy/paste the symbols here and remove this section before submitting.*

**This pipe-connecting game can be modeled as a \*\*graph-based puzzle\*\* with nodes (pipe pieces) and edges (connections between pipes). Here’s a way to approach modeling the game formally:**

**### 1. \*\*Define the Grid and Pipe Types\*\***

**- \*\*Grid\*\*: Represent the 3x3 grid as a matrix where each cell contains one pipe.**

**- \*\*Pipe Types\*\*: Assume three types of pipes:**

**- \*\*Straight Pipe\*\* (`|` or `—`): Connects either horizontally or vertically.**

**- \*\*Corner Pipe\*\* (`┌`, `┐`, `└`, `┘`): Connects two sides at a 90-degree angle.**

**- \*\*T-Connector\*\* (`┬`, `┤`, `┴`, `├`): Connects three sides (useful for intersections).**

**- \*\*Start and End\*\*: Define the start at the top-left corner `(0, 0)` and the end at the bottom-right corner `(2, 2)`.**

**### 2. \*\*Model the Rotations\*\***

**- Each pipe type can have multiple orientations. For instance:**

**- \*\*Straight Pipe\*\*: Can be rotated 0° (horizontal) or 90° (vertical).**

**- \*\*Corner Pipe\*\*: Has four orientations: 0°, 90°, 180°, and 270°.**

**- \*\*T-Connector\*\*: Has four orientations.**

**- Define a function or rule to model how each piece rotates and the resulting connections.**

**### 3. \*\*Define the Connectivity Constraints\*\***

**- Each cell in the grid connects to neighboring cells based on the orientation of the pipe within it.**

**- Use propositions for each cell to indicate if it connects to neighboring cells:**

**- `Connects(cell1, cell2)`: True if the pipe in `cell1` connects to `cell2`.**

**- Connectivity rules are based on the orientation of each pipe:**

**- For example, if a straight pipe is horizontal in `(0, 0)`, `Connects((0, 0), (0, 1))` and `Connects((0, 1), (0, 0))` should be true.**

**### 4. \*\*Objective Constraints\*\***

**- Define constraints to ensure the grid is connected from start to end:**

**- \*\*Path Continuity\*\*: Starting at `(0, 0)`, every cell along a path must have an active connection leading to the next cell until reaching `(2, 2)`.**

**- \*\*Full Coverage\*\*: Every pipe in the grid must be part of a continuous path from the start to the end.**

**- \*\*No Loose Ends\*\*: Each pipe should connect to neighboring pipes without leaving any dangling ends (unconnected pipe openings).**

**### 5. \*\*Rotations as Decisions\*\***

**- Represent each pipe’s rotation as a decision variable:**

**- For example, `Rotate(cell, angle)` where `angle` is one of the allowed rotations (0°, 90°, etc.).**

**- The objective of the game is to assign a rotation to each pipe such that all connectivity constraints are satisfied.**

**### 6. \*\*Model Validation\*\***

**- Use a \*\*graph traversal\*\* (like Depth-First Search or Breadth-First Search) to check if there’s a valid path from `(0, 0)` to `(2, 2)` that includes every cell.**

**- Verify that all cells connect as required without any open connections.**

**### 7. \*\*Example in Propositional Logic\*\***

**- \*\*Variables\*\*: For each cell `(i, j)`, define a rotation variable and connectivity variables, e.g., `Rotate(cell, angle)` and `Connects(cell1, cell2)`.**

**- \*\*Constraints\*\*:**

**- `Rotate((i, j), 90°) → (Connects((i, j), (i, j+1)) ∧ Connects((i, j+1), (i, j)))`: If a cell is rotated 90°, it must connect to its right neighbor if it’s a straight or corner pipe.**

**- `PathExists((0,0), (2,2))`: Ensure there’s a continuous path from the start to the end.**

**This model allows the game to be simulated logically by assigning rotations and testing connectivity across the grid, fulfilling the objective of connecting all pipes from start to finish.**

**To model a solution where pipe orientations are set non-sequentially, we can use a \*\*constraint satisfaction approach\*\* rather than a step-by-step sequence. This allows us to specify rotations for specific pipes without enforcing an order, ensuring that all pipes meet the connectivity requirements regardless of the order in which we assign orientations.**

**Here's how we can approach this:**

**### 1. \*\*Define Each Pipe’s Possible Orientations as Independent Variables\*\***

**- For each cell `(i, j)` in the grid, define a variable `Orientation(i, j)` that represents the pipe’s orientation. The possible values of this variable depend on the type of pipe:**

**- \*\*Straight Pipe\*\*: `{0°, 90°}`**

**- \*\*Corner Pipe\*\*: `{0°, 90°, 180°, 270°}`**

**- \*\*T-Connector\*\*: `{0°, 90°, 180°, 270°}`**

**- These orientations do not rely on any other cell's rotation and can be set in any order.**

**### 2. \*\*Set Connectivity Constraints Across the Entire Grid\*\***

**- Define connectivity constraints between neighboring cells to ensure that the orientations chosen result in a continuous path from the start `(0,0)` to the end `(2,2)`.**

**- Example of connectivity constraints:**

**- If cell `(i, j)` has a pipe facing \*\*east\*\* and its neighbor `(i, j+1)` has a compatible pipe facing \*\*west\*\*, they should connect:**

**- `Orientation(i, j) = east ∧ Orientation(i, j+1) = west → Connects((i, j), (i, j+1)) ∧ Connects((i, j+1), (i, j))`**

**- These constraints apply to each neighboring pair based on their pipe type and orientation.**

**### 3. \*\*Model the Path Requirement as a Global Constraint\*\***

**- Define a \*\*path continuity constraint\*\* to ensure a complete connection from `(0,0)` to `(2,2)`. This constraint ensures that all pipes form a connected path without requiring a specific order of orientation assignments:**

**- Let `PathExists(start, end)` represent a path from the start to the end.**

**- This constraint can be checked through a \*\*graph traversal\*\* (e.g., DFS or BFS) that verifies if there is a complete path from `(0,0)` to `(2,2)` using the current orientations of each pipe.**

**### 4. \*\*Add Specific Orientation Constraints for Selected Pipes\*\***

**- If specific orientations are required for certain pipes, you can set constraints directly on those cells without depending on the order in which they are processed.**

**- For instance, if:**

**- The third pipe must face east: `Orientation(0, 2) = east`**

**- The fifth pipe must face north: `Orientation(1, 1) = north`**

**- These constraints are \*\*independent\*\* and don’t enforce any rotation sequence.**

**### 5. \*\*Constraint Satisfaction Problem (CSP) Setup\*\***

**- With the above constraints, the model becomes a \*\*CSP\*\* where:**

**- Variables: `Orientation(i, j)` for each cell `(i, j)`**

**- Domains: Possible orientations for each pipe type**

**- Constraints:**

**- Connectivity constraints to ensure neighboring pipes connect properly**

**- Path requirement to ensure a full path from start to end**

**- Specific orientation constraints on selected cells as needed**

**- The CSP solver will assign values to each `Orientation(i, j)` to satisfy all constraints simultaneously, regardless of order.**

**### 6. \*\*Path Validation\*\***

**- After assigning orientations to all pipes, validate the solution by checking that:**

**- Each pipe connects correctly to its neighbors.**

**- There is a valid path from `(0,0)` to `(2,2)` using a graph traversal to confirm connectivity.**

**### Summary**

**Using this approach, the model allows for the orientation of specific pipes without needing a strict sequential order. Each pipe’s orientation can be set independently, and the global connectivity and path requirements ensure that the final solution forms a connected path from start to end. This structure is ideal for solving the puzzle using a CSP solver or a similar non-sequential logic framework.**