# Project Summary

Short summary of the project setting.

This dungeon crawler project aims to model the creation of randomized layout of rooms, mimicking similar systems in popular roguelike/roguelite video games such as The Binding of Isaac and Hades.

For every dungeon layout, there are 7 rooms to be placed on a 13x13 grid while adhering to the constraints outlined below. There are two types of special rooms: a starting room, (where the theoretical player would enter the dungeon, which is always placed in the center of the grid) and an ending room (where the player would complete the dungeon). The other five rooms are regular rooms with no special properties.

# Propositions

List of the propositions used in the model, and their (English) interpretation.

* sufficient\_length (i,j): This is true when the minimum amount of moves to get from Start tile **i** to End tile **j** is 4 or more.
* reachable(i, j, n): This is true when tile i can reach tile j in n steps.
* special\_adjacency (k): This is true when a special tile **k** is adjacent to exactly 1 other tile. This means that there is only one connection to the start and end tiles.
* connected (i, j, d): This is true when tiles **i** and **j** are adjacent in the direction **d**, relative to tile **i**. **d** can be one of four cardinal directions (N, E, S, W)
* occupied(loc, t): This is true when the location **loc** on the grid relative to the start tile is occupied by the tile **t**.
* 2\_connections(t): This is true when tile **t** has exactly two adjacent tiles.
* all\_tiles\_placed(m): This is true when 13 tiles exist in the dungeon **m**.

## Finite Domain: Room Types

The following propositions are used to identify the type of each tile. If one of these propositions are satisfied, none of the others are satisfied. This way, we ensure that every room has exactly one room type, and that there are no duplicates of the special rooms. If none of these propositions are satisfied, the room is a regular room.

* is\_start(i): This is true when tile **i** is the starting tile.
* is\_end(i): This is true when tile **i** is the end tile.
* ¬is\_start(i) ∧ ¬is\_end(i): This is true when the tile **i** is a regular tile.

# Constraints

### Constraints on the Dungeon Layout

#### Special Tile Uniqueness

* + Constraint: ¬(is\_start(t₁) ∧ is\_start(t₂)), where t₁ and t₂ are arbitrary unique tiles.
  + Constraint: ¬(is\_end(t₁) ∧ is\_end(t₂)), where t₁ and t₂ are arbitrary unique tiles.
  + Ensures that each special tile type occurrence is unique.

#### Adjacency

* + Constraint: reachable(t₁, t₂, 1) ∧ connected(t₁, t₂, d), where t₁ and t₂ are arbitrary unique tiles and d is a cardinal direction.
  + Each tile must connect to at least one other tile, ensuring a continuous path across all tiles.

#### Single-Occupancy

* + Constraint: t₁ ∧ t₂ ¬occupied(loc, t1) ∧ ¬occupied(loc, t2), where t₁ and t₂ are arbitrary unique tiles, and loc is an arbitrary location
  + Constraint: ¬(occupied(loc, t₁) ∧ occupied(loc, t₂)), where t₁ and t₂ are arbitrary unique tiles and loc is a grid location relative to the start tile.
  + Ensures no overlapping tile placements by comparing occupancy.

#### Valid Path Length

* + Constraint: sufficient\_length(s, e) ¬(reachable(s, e, 1) ∧ reachable(s, e, 2) ∧ reachable(s, e, 3)), where s is the start tile, and e is the end tile.
  + Enforces a minimum of four moves from the start tile to the end tile.

#### Reachability

* + Constraint: t ∧ reachable(s, t, n), where t is an arbitrary tile, s is the start tile, and n is an arbitrary number of steps.
  + Constraint: reachable(s, t1, n1) ∧ reachable(s, t2, n2) ∧ … ∧ reachable(s, t12, n12), where t1 - t12 are unique arbitrary tiles, s is the start tile, and n1 - n12 are unique arbitrary distances
  + All tiles must connect to the starting tile, ensuring a fully traversable dungeon.

#### Connection Count

* + Constraint: ¬(is\_start(t) V is\_end(t)) 2\_connections(t), where t is an arbitrary non-special tile.
  + Constraint: 2\_connections(t1) (connected(t1, t2, d1) ∧ connected(t1, t3, d2)) where t1 to t3 are arbitrary tiles and d1 to d2 are different cardinal directions.
  + Non-special tiles must have exactly two connections, forming simple paths without branching.

#### Special Tile Connections

* + Constraint: special\_adjacency(s) connected(s, t, d) where s is the start tile, t is an arbitrary tile and d is a cardinal direction.
  + Constraint: special\_adjacency(e) connected(e, t, d), where e is the end tile, t is an arbitrary tile and d is a cardinal direction.
  + Special tiles are limited to adjacency with only one other tile, therefore only have one adjacent connection.

# Model Exploration

In this section, we highlight some of the ideas, struggles, and successes of implementing our model.  
Some of these ideas are from early stages of our project, and may have been replaced by another solution in our final model.

## Using a 2D list to store the room locations.

We decided to store our tiles in a 2D list. This way, we can easily identify the locations of each tile using (x, y) coordinates and avoid traversing a long link of nodes.

Our model ensures that there are only 13 nodes. If all the tiles are in a straight line, it will be 13 tiles long from end to end. So, we decided to use a 25x25 2D list and place our starting tile directly in the center (index [12][12]). This way, we will never exceed the bounds of the array.

## Creating objects for tiles vs. using True / False

As we started implementing our model in Python, we needed a way to store our tiles. Based on our previous idea of using a 25x25 2D list, we knew we would store our tiles in there. However, we needed to determine what data to store in the list.

Consequently, we devised two approaches: an object-oriented approach where each element in the list is a Tile object, or a boolean system where each element is a True or False value. We have included our reasoning in the tables below:

### *Table 1: Object-Oriented Tile System*

| Pros | Cons |
| --- | --- |
| * Can store all necessary information inside the objects (a dictionary of neighboring nodes, the tile’s position, room type, etc.) * Only needs one 2D list | * Encapsulating tile data into a singular object might make the model difficult to navigate. |

### *Table 2: Object-Oriented Tile System*

| Pros | Cons |
| --- | --- |
| * Keeps the data concise (“is there a room here?” – that’s the fundamental question we’re asking) * Can identify the neighbors by simple math (check adjacent tiles using current tile’s (x,y) position) | * There’s no easy way to store the room type with the tile. We would need to use parallel arrays, thus overcomplicating the problem. |

Given these factors, we decided that an object-oriented approach would be best.

## First Bug: Assigning tiles to all positions

When creating our 2D list of tiles, one of our first approaches was to populate every element in the 25x25 list with a tile, either being the starting room or a regular room. However, since we only have 13 tiles to place, this immediately violates that constraint.  
  
 To find a solution, we need to find a way to determine every valid arrangement of tiles, keeping in mind the constraint that all tiles must be connected. So, randomly placing each tile is not a valid solution. Instead, we need to find a way to continuously place tiles by branching out from the starting tile.

One possible implementation is to start from the start tile, and branch out in each adjacent direction, then continuing to branch out for every possible combination of tiles. However, this will likely result in a combinatorial explosion, so we are seeking guidance for how to best approach this problem.

## Preventing Islands of Tiles

In our project proposal, we imposed a constraint that every tile must connect to at least one tile. We thought that by doing so, every tile would connect to each other. However, Professor Muise pointed out that this constraint does not guarantee that every tile can reach one another.

For instance, if 11 tiles connect to each other, and two tiles connect to each other, but there is no connection between the 11 tiles and the two tiles, then this would break the system. He suggested instead that we should create a constraint where for every tile, you can reach all other tiles from that tile.

To accomplish this, we need to use our **reachable**(**i, j, d)** proposition, indicating that from tile **i**, we can reach tile **j** in **d** steps. While this is possible in Python, this is difficult to prove in Jape, since we cannot use numbers in our available forms of logic. As such, we are seeking guidance on how to implement this effectively.

## 

## Changing Map Dimensions and Number of Tiles to Decrease Simulation Time

Initially, our model placed 13 tiles on a 25x25 grid, applying the propositions and constraints for each tile. We were confident this would generate some interesting solutions, but we severely underestimated the required number of computations. When we tried to run a basic version of our model, it took approximately 3 minutes to generate a single solution.

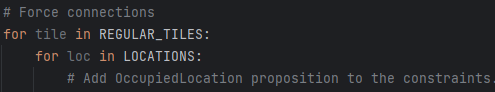
Clearly, our current approach was unsustainable and highly inefficient. So, we looked for ways to reduce the number of computations for each solution. Using fewer tiles, we dramatically reduced the number of computations. So, we settled on using 7 tiles. Then, because our grid size depends on our number of tiles, we could decrease the grid size too. The length and width of the grid are represented by **g = 2r - 1**, where **r** is the number of tiles in the dungeon. With a grid length/width of **g**, there are **g2** possible locations. So, by reducing **r** from 13 to 7, the number of possible locations decreased from 625 to 169, causing the model to run much faster.

## Removing the Trap and Treasure Tiles

We had many ideas to make our project more complex. This included two more special tiles, “trap” and “treasure”, which each had their own propositions and constraints. Once we started our Python implementation, we quickly realized the scope of our ideas. In the interest of time, we decided to downsize and create a simpler, working model. Consequently, we removed the trap and treasure tiles from our model.

## Tile Constraints Getting Overwritten

When implementing the constraint that two tiles cannot share the same location, we quickly discovered a flaw in our logic. Referring to previous projects, many used list comprehension in the **constraint.add\_exactly\_one(E,** proposition**)** function. We did not fully understand how the function worked, so we figured adding constraints could work using a nested for-loop instead of list comprehension. This broke our model, as there could be multiple tiles at the same location.

  
*This was our initial approach; however, it would add exactly one constraint for every (tile, loc) pair, which was not our desired result.*  
  
 We arrived at our solution after reading the function’s documentation. The **add\_exactly\_one** function adds exactly one proposition from a *list* of propositions into the theory’s constraints. So, we realized that the list comprehension was vital to the function, and we could not replace it with a nested for loop. We attached the fixed code in the image below.

  
*After reading the documentation, we realized we needed to pass in an array of propositions.*

## A Happy Discovery: Some Propositions Write Themselves!

As we added more constraints to our model, we discovered that some constraints forced certain propositions to be true, even if we did not define the proposition in our model! For instance, one of our initial goals for the model was to ensure that the start and end tiles are at least four tiles apart (i.e, **sufficient\_length(start, end)** must evaluate to T).

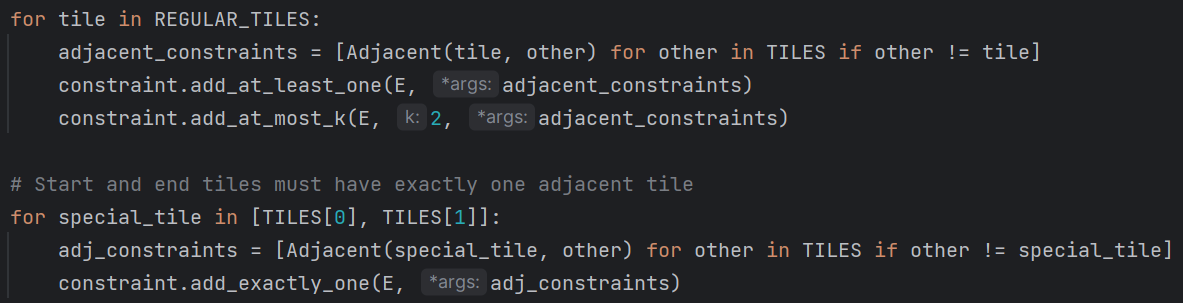
Before implementing this constraint, we implemented our adjacency constraints: every regular tile must have exactly two connections, and the start and end tile must have exactly one connection, respectively. Then, we added the constraint that there must exist a path between all tiles. When we ran our model, we saw that whenever these constraints were satisfied, **sufficient\_length(**start, end**)** must be true, even though we did not include it as a constraint!

Each of the constraints contributes to this phenomenon. First, if every regular tile has two connections, then the dungeon cannot end on a regular tile. If only this constraint was applied, it would create a loop of regular tiles. Next, if the start and end tile must have exactly one connection, then they can connect to the regular tiles, breaking the loop. However, this allows islands to occur. There can exist an island with the start tile adjacent to the end tile, with all regular tiles on another island – since start is adjacent to end, **sufficient\_length(start, end)** is false. To fix this, apply the constraint that there must exist a path between all tiles. By doing so, this removes the islands and the loop of regular tiles. By the first constraint, all regular tiles must have exactly two connections. Then, since **start** and **end** must have exactly one connection, they will connect to a regular tile on opposite ends of the path. There are more than four regular tiles, so **start** and **end** will have at least four tiles between them.

We discovered that by implementing these three constraints, the start tile and end tile must always be at least 4 tiles apart, so **sufficient\_length(start, end)** must hold. Therefore, we did not need to define this proposition in our model.

## Fixing Islands of Tiles

As we explored solutions to our “islands of tiles” bug, we tried to ensure that every regular tile had exactly two connections, and that the **start** and **end** tile must have exactly one connection. We discovered that there was no **constraint.add\_exactly\_k** function, and we were unsure how to progress. We settled on the following code:



This guaranteed that every tile must have between 1 and 2 (inclusive) connections. However, when we tested this implementation, our grid still contained islands of tiles. To fix this, we implemented an algorithm to ensure a path exists between every tile. First, we created a list of visited tiles. Then, we iterated through all tiles and added a constraint that the tile must be adjacent to an unvisited tile. By doing this for all tiles, we ensured that there exists a path between all tiles, and that no tiles would be on an island.

# Jape Proofs

##### Sequent 1 | Reachability when Placed

We can prove that if all tiles are placed, all tiles are reachable from the start tile.

For this proof, we will represent each tile as Tn from T0 to T12 where n = the number of moves required to get to the tile from the start tile. Each tile atom will be true if placed and false if not placed. We will also represent the constraint as an atom:

reachable(T0, Tn, n) = Rn

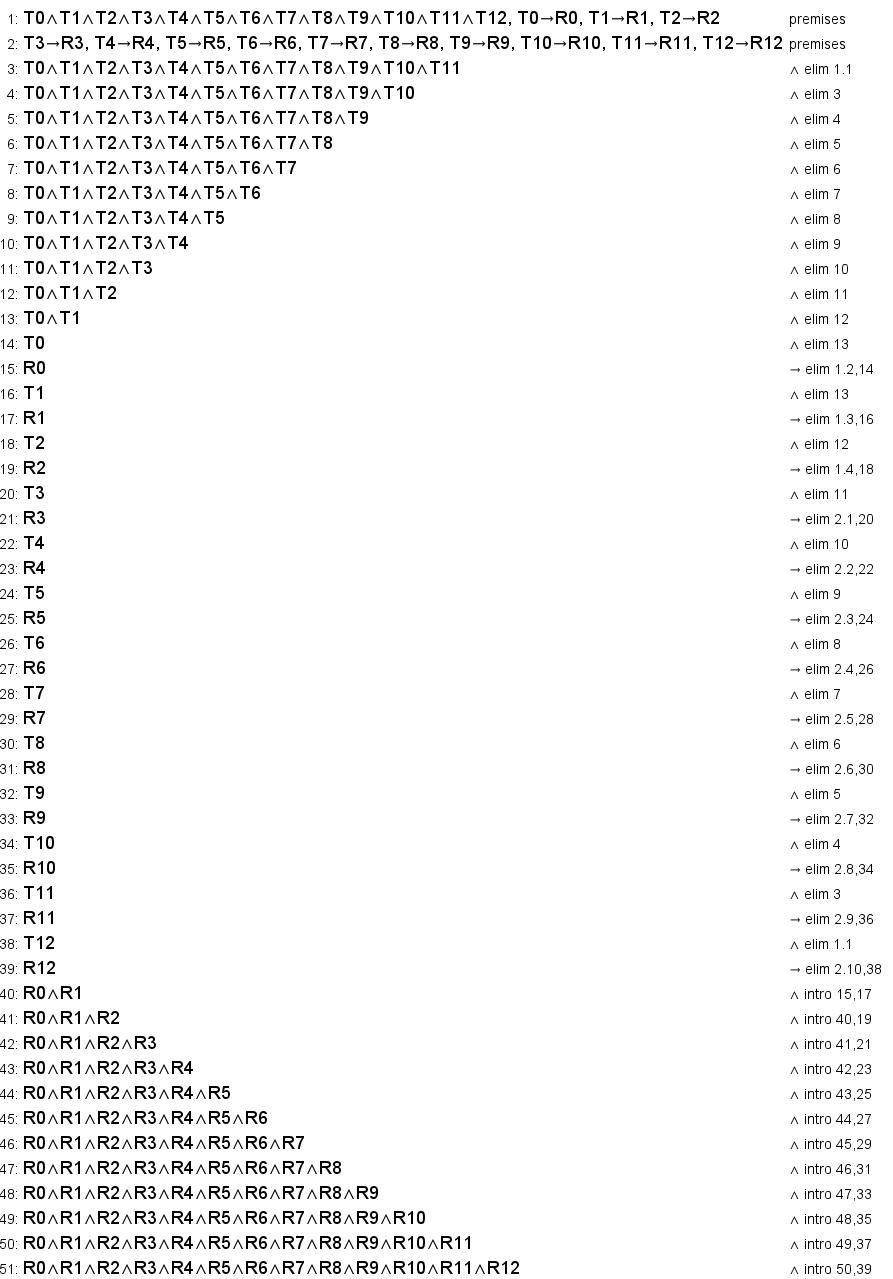
Premises:

Simplified: Tn, Tn Rn  
Expanded: T0 ∧ T1 ∧ T2 ∧ T3 ∧ T4 ∧ T5 ∧ T6 ∧ T7 ∧ T8 ∧ T9 ∧ T10 ∧ T11 ∧ T12, T0 R0, T1 R1, T2 R2,  
T3 R3, T4 R4, T5 R5, T6 R6, T7 R7, T8 R8, T9 R9, T10 R10, T11 R11, T12 R12

Conclusion:

Simplified: Rn  
Expanded: R0 ∧ R1 ∧ R2 ∧ R3 ∧ R4 ∧ R5 ∧ R6 ∧ R7 ∧ R8 ∧ R9 ∧ R10 ∧ R11 ∧ R12

Jape Proof: ***listed on next page***



##### Sequent 2 | Regular Tile Adjacencies

We can prove that we cannot have a regular tile with 3 adjacencies given the definition of a regular tile and the fact that no tiles may overlap (two connections cannot be in the same direction).

For this proof we will give the tile we’re using to prove the index number 0, and the 3 adjacent tiles index numbers 1 to 3. For directions we will represent them as N = 0, E = 1, S = 2, W = 3. We will also represent the constraints as atoms:

(¬is\_start(t0) ∧ ¬is\_end(t0)) = R

2\_connections(t0) = S

connected(t0, tx d) = Cxd

Premises:

English: Regular tiles have two connections, A tile with two connections must fit at least one situation where one of three adjacencies is false.

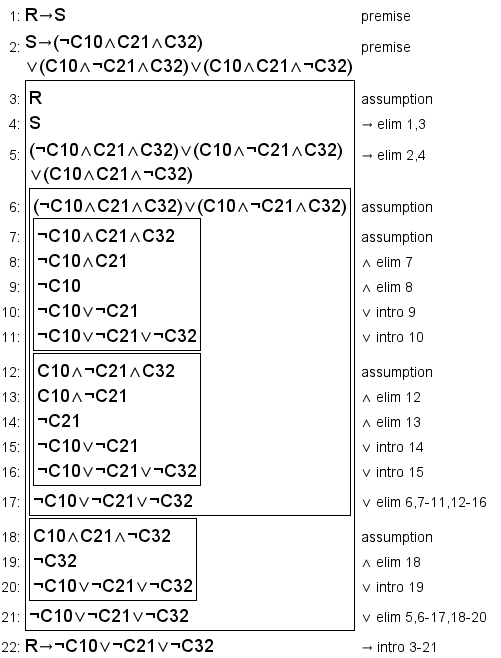
Logic: R S, S (¬C10 ∧ C21 ∧ C32) (C10 ∧ ¬C21 ∧ C32) (C10 ∧ C21 ∧ ¬C32),

Conclusion:

English: Regular tiles have at least one false adjacency given three possible adjacencies.

Logic: R ¬C10 ¬C21 ¬C32

Jape Proof: ***listed on next page***



##### 

##### Sequent 3 | Reachability when Connected

We can prove that the nth tile from the starting tile is reachable in n steps when each tile is connected to the previous tile (until the starting tile is reached).

For this proof we will label each tile with indexes from T0 to T12. We will also represent the constraints as atoms, where T0 is always assumed to be the start tile, T12 is always assumed to be the end tile, and d is always assumed to be a valid direction for connecting tiles according to our model’s constraints:

connected(Tx, Ty, d) = Cxy

reachable(T0, Tn, n) = Rn

Premises:

English: Given our path from the start tile to the end tile, the nth tile placed from the starting tile is reachable from the start tile in n steps.

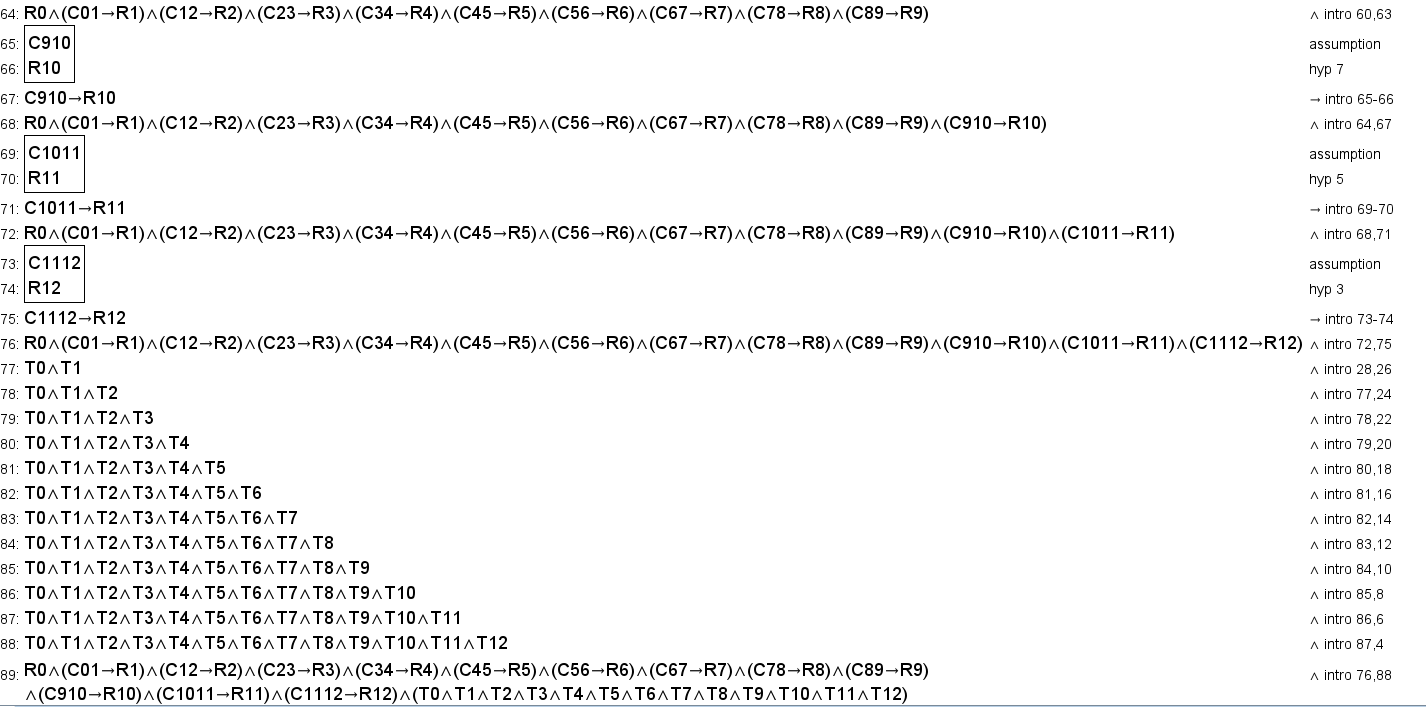
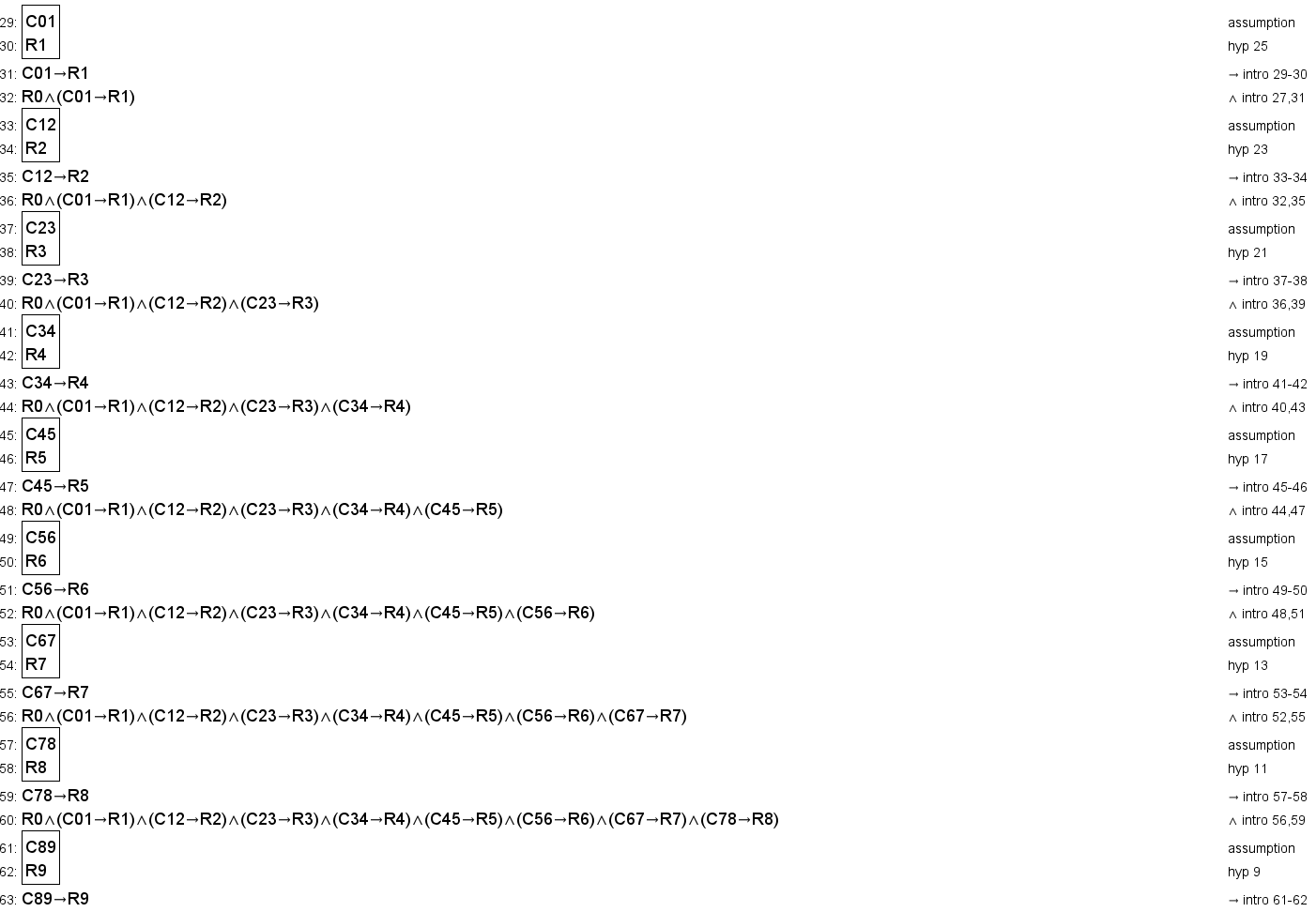
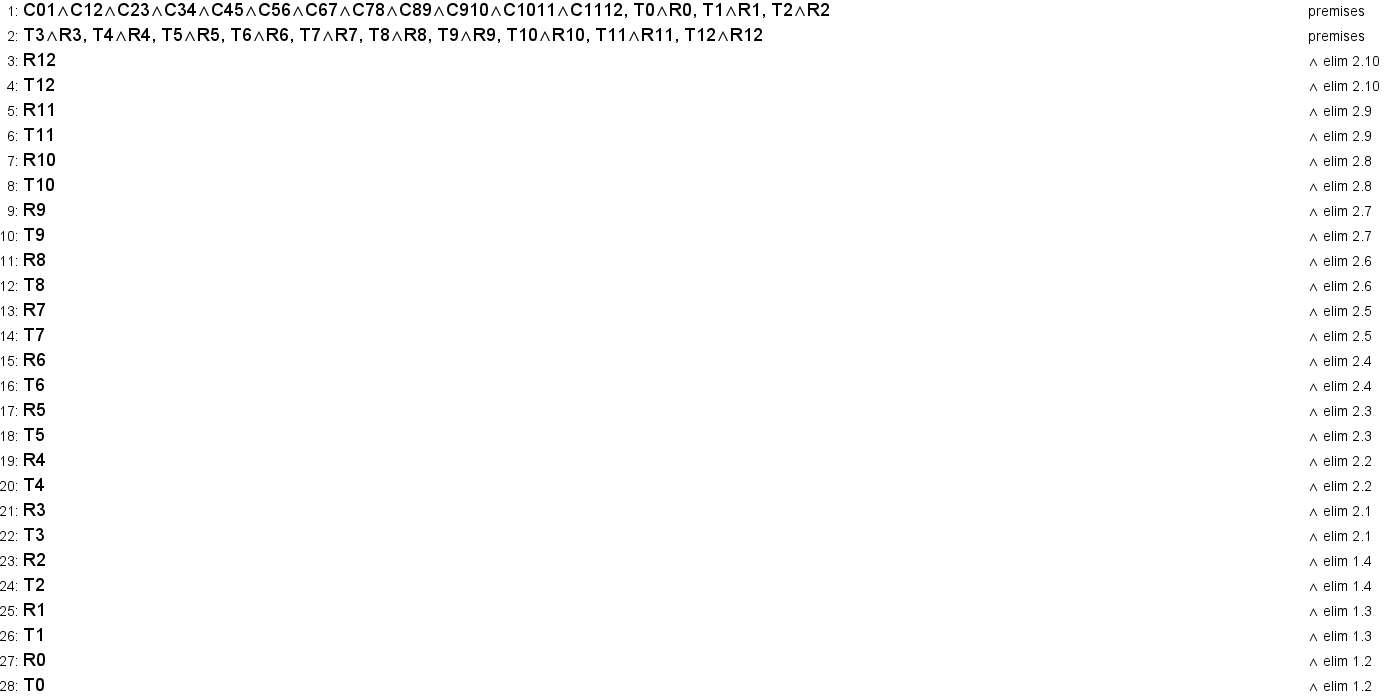
Logic: C01 ∧ C12 ∧ C23 ∧ C34 ∧ C45 ∧ C56 ∧ C67 ∧ C78 ∧ C89 ∧ C910 ∧ C1011 ∧ C1112, T0 ∧ R0, T1 ∧ R1, T2 ∧ R2, T3 ∧ R3, T4 ∧ R4, T5 ∧ R5, T6 ∧ R6, T7 ∧ R7, T8 ∧ R8, T9 ∧ R9, T10 ∧ R10, T11 ∧ R11, T12 ∧ R12

Conclusion:

English: When all tiles are placed and tile n is connected to tile n-1, tile n is reachable in n steps.

Logic: (T0 ∧ T1 ∧ T2 ∧ T3 ∧ T4 ∧ T5 ∧ T6 ∧ T7 ∧ T8 ∧ T9 ∧ T10 ∧ T11 ∧ T12) ∧ R0 ∧ (C01 R1) ∧ (C12 R2) ∧ (C23 R3) ∧ (C34 R4) ∧ (C45 R5) ∧ (C56 R6) ∧ (C67 R7) ∧ (C78 R8) ∧ (C89 R9) ∧ (C910 R10) ∧ (C1011 R11) ∧ (C1112 R12)

Jape Proof: ***listed on next page***



# First-Order Extension

For this extension, we will rewrite each applicable constraint in predicate logic, as our propositions can already be applied effectively:

#### Special Tile Uniqueness

* + ∀m. ∃t₁. ∃t₂. (is\_start(t₁) ∧ is\_end(t₂)), where m is a dungeon and t₁ and t₂ are tiles. (Difficult to fully capture the uniqueness aspect, but this remains true.)

#### Adjacency

* + ∀t₁. ∃t₂. ∃d. (reachable(t₁, t₂, 1) ∧ connected(t₁, t₂, d)), where t₁ and t₂ are tiles and d is a cardinal direction. (Sort of ties in more to Connection Count than Adjacency, but still relevant.)

#### Single-Occupancy

* + ∀t₁. ¬∃t₂. occupied(t1,t2), where t1 and t2 are tiles.
  + ∀loc. ¬∃t₁. ¬∃t₂. (occupied(loc, t₁) ∧ occupied(loc, t₂)), where t₁ and t₂ are tiles and loc is a grid location relative to the start tile.
  + ∀loc. ∃t. occupied(loc, t), where t is a tile and loc is a grid location relative to the start tile. (Not part of the original constraints, but helps in understanding our first-order extension.)

#### Valid Path Length

* + ∀m. ∃s. ∃e. sufficient\_length(s, e), where m is a dungeon, s is the start tile, and e is the end tile. (Difficult to fully translate the implication, but this is still true.)

#### Reachability

* + ∀t. ∃n. reachable(s, t, n), where t is a tile, s is the start tile, and n is an arbitrary number of steps. (Fully encapsulates both constraints with this one predicate logic statement.)

#### Connection Count

* + ∀t. 2\_connections(t), where t is a non-special tile.

#### Special Tile Connections

* + ∀s. ∃t. ∃d. (special\_adjacency(s) connected(s, t, d)), where s is the start tile, t is a tile and d is a cardinal direction.
  + ∀e. ∃t. ∃d. (special\_adjacency(e) connected(e, t, d)), where e is the end tile, t is a tile and d is a cardinal direction.

# 

# 

# 

# Results

Using propositional logic, we created a model to create map layouts for dungeon crawlers. By exploring our model in Python, we identified propositions that were already implied by existing constraints, allowing us to optimize our model. At the same time, we had to reduce the scope of our problem, both in the interest of time and to run our model efficiently – this proved vital to our success.

To interact with our model, we created a display grid that represents the level map. Every ‘X’ represents an unoccupied location on the map. The ‘S’ tile marks the location of the start tile. In our model, we ensured that the start tile is always at the center of the map. Then, the ‘E’ tile marks the location of the end tile, and the ‘R’ tiles show the location of the regular tiles.

  
*An example map of tiles generated by our model.*

Using our model, we created randomized paths of tiles, generating exciting level maps for a dungeon crawler game.