# 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 12 rooms that will be randomly placed while adhering to the constraints outlined below. In addition to the 8 normal rooms, there is one starting room, (where the theoretical player would enter the dungeon), one ending room (where the player would complete the dungeon), one treasure room (where the player would find some sort of rare loot), and one trap room (where the player would encounter a challenging obstacle of some sort). Each of these four special rooms have their own special constraints to follow, either adding to or altering constraints that normal rooms follow.

# 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.
* treasure (i,j): This is true when the treasure tile **i** and the end tile **j** are on opposite ends of the map. This means that the distance between **i** and **j** is greater than or equal to the distance between **i** and every other tile.
* reachable(i, j, n): This is true when tile i can reach tile j in n steps.
* end\_adjacency (i, k): This is true when the Treasure tile **i** and End tile **k** are adjacent to exactly 1 other tile, respectively. This means that there is only one connection to treasure tile **i**, and one connection to end tile **k**
* 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 is occupied by the tile **t**.
* 2\_connections(t): This is true when tile **t** has exactly two adjacent tiles.

## 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\_trap(i): This is true when tile **i** is a booby trap tile.
* is\_treasure(i): This is true when tile **i** is a treasure tile.
* 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.

# Constraints

List of constraint types used in the model and their (English) interpretation. You only need to provide one example for each constraint type: e.g., if you have constraints saying “cars have one colour assigned” in a car configuration setting, then you only need to show the constraints for a single car. Essentially, we want to see the pattern for all of the types of constraints, and not every constraint enumerated.

### Constraints on the Dungeon Layout

* Uniqueness Constraints
  + Constraint: ¬(is\_treasure(t₁) ∧ is\_treasure(t₂)), where t₁ and t₂ are unique tiles.
  + Ensures that each special tile type (treasure, trap, start, and end) is unique, preventing duplicates of any special type.
* Adjacency Constraints
  + Constraint: ∀t₁ ∃t₂ ∃d, connected(t₁, t₂, d).
  + Each tile must connect to at least one other tile, ensuring a continuous path across all tiles.
* Placement Constraints
  + Constraint: ∀t ∃loc, occupied(loc, t).
  + Every tile must be assigned a location on the grid, completing the dungeon layout.
* Single-Occupancy Constraints
  + Constraint: ∀loc, t₁ ≠ t₂ → ¬(occupied(loc, t₁) ∧ occupied(loc, t₂)).
  + Ensures that each grid location can hold only one tile, preventing overlaps.
* Path Length Constraint
  + Constraint: sufficient\_length(start, end).
  + Enforces a minimum of four moves from the start tile to the end tile.
* Limited Adjacency for Special Tiles
  + Constraint: end\_adjacency(treasure, end).
  + Treasure and end tiles are limited to adjacency with only one other tile, placing them at the "edges" of paths
* Relative Distance Constraint
  + Constraint: treasure(t, end).
  + The distance from the treasure tile to the end tile must be greater than or equal to the distance from the end tile to any other tile.
* Reachability Constraints
  + Constraint: ∀tiles, reachable(tile, start) ∧ reachable(tile, end).
  + All tiles must connect to both start and end tiles, ensuring a fully traversable dungeon.
* Connection Count for Non-Special Tiles
  + Constraint: ∀non\_special\_tile, 2\_connections(tile).
  + Non-special tiles must have exactly two connections, forming simple paths without branching.
* Grid Position for Start Tile
  + Constraint: start\_tile\_location = (13, 13) on a 25x25 grid.
  + The start tile is fixed at the center of the grid.
* Boundary Constraints
  + Constraint: ∀tiles, loc\_x ∈ [0, 24] ∧ loc\_y ∈ [0, 24].
  + All tile positions must fall within the grid’s 25x25 bounds.
* Total Tile Count
  + Constraint: total\_tiles = 13.
  + Fixes the total count of tiles in the dungeon.
* Special and Regular Tile Classification
  + Constraints:
    - regular\_tiles = 9, no special properties
    - trap\_tiles = 1
    - treasure\_tiles = 1
    - start\_tiles = 1
    - end\_tiles = 1
  + Ensures a mix of 9 regular tiles and 4 unique special tiles (Trap, Treasure, Start, and End) totaling 13 tiles.

# Model Exploration

List all the ways that you have explored your model – not only the final version, but intermediate versions as well. See (C3) in the project description for ideas.

## 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.

# Jape Proof Ideas (Paul)

**Sequent 1**

SufficientLength(i, j, StepFind(i, j)) ∧ ∀k (SufficientLength(k, j) → (¬MaxMoves(k, j) ∨ MaxMoves(i, j))) ⊢ IsTreasure(i, j)

**Left Side of the Sequent**

- SufficientLength(i, j, StepFind(i, j)): This is true when the minimum number of moves to get from Start tile i to End tile j is 4 or more and there is a non-infinite integer of legal steps separating the two tiles.

- ∀k (SufficientLength(k, j, StepFind(i,j)) → (¬MaxMoves(k, j) ∨ MaxMoves(i, j))): This part universally quantifies over all tiles k:

- SufficientLength(k, j): Indicates that tile k has a valid path to End tile j.

- ¬MaxMoves(k, j): Indicates that tile k does not have the maximum number of moves to reach End tile j.

- MaxMoves(i, j): States that tile i has more moves than tile k to reach End tile j.

**Right Side of the Sequent**

- IsTreasure(i, j): This is the conclusion we want to reach. If the conditions on the left side are satisfied, it asserts that tile i is a Treasure tile that has the maximum number of moves to reach End tile j.

**Explanation of Symbols i, j, and k**

- i: Represents the current tile being evaluated in the context of determining whether it qualifies as a Treasure tile. This tile is checked against other tiles to see if it has the maximum number of moves to reach the End tile.

- j: Represents the End tile in the context of the game or grid. It is the destination tile that all evaluated paths lead to. The propositions are concerned with how many moves it takes to reach this specific End tile from various Start tiles.

- k: Represents other tiles that are evaluated during the process. For each k, the sequent checks if the moves from tile k to End tile j are less than or equal to the moves from tile i to the same End tile j. The variable k is used to assess all other tiles that may have valid paths to the End tile.

**Sequent 2**

(adjacent\_left(i) ∧ connected(adjacent\_tile\_left(i), j)) ∨ (adjacent\_right(i) ∧ connected(adjacent\_tile\_right(i), j)) ∨ (adjacent\_up(i) ∧ connected(adjacent\_tile\_up(i), j)) ∨ (adjacent\_down(i) ∧ connected(adjacent\_tile\_down(i), j)) ⊢ StepFind(i, j)

**Left Side of the Sequent:**

- adjacent\_left(i): Asserts that tile i has an adjacent tile on its left.

- connected(adjacent\_tile\_left(i), j): Indicates that there is a path from the tile to the left of i to tile j.

- adjacent\_right(i): Asserts that tile i has an adjacent tile on its right.

- connected(adjacent\_tile\_right(i), j): Indicates that there is a path from the tile to the right of i to tile j.

- adjacent\_up(i): Asserts that tile i has an adjacent tile above it.

- connected(adjacent\_tile\_up(i), j): Indicates that there is a path from the tile above i to tile j.

- adjacent\_down(i): Asserts that tile i has an adjacent tile below it.

- connected(adjacent\_tile\_down(i), j): Indicates that there is a path from the tile below i to tile j.

**Right Side of the Sequent:**

- StepFind(i, j): This is the conclusion we want to establish, indicating that there is a valid path from tile i to tile j.

**Explanation of Symbols**

- i: Represents the current tile being evaluated for connectivity.

- j: Represents the End tile that we want to determine if it is reachable from tile i.

# Requested Feedback

Provide 2-3 questions you’d like the TA’s and other students to comment on.

1. How can we use numbers inside our propositions/constraints?

* For Jape proofs, we cannot include numbers, so how can we go about this?

1. How can we **explore every combination** **and connection** when placing our tiles in Python?

* We want to avoid a massive number of computations (combination explosion)

1. Do we need to prove all of our propositions and constraints in Jape?

# 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!**

We are already using plenty of predicate logic, so we have already done this.

# Useful Notation

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