Group 11: Treasuring Land

Project Summary

“Treasure Land” is a brave adventure game. The player has to follow the clues to find the treasure. However, in the process of searching for the treasure, there will be many difficulties, such as marsh, weather, or getting hurt for unknown reasons, which will lead to death. On the other hand, in the process of searching for the treasure, there will also be things happening in the favor of the player. For example, you will find an artifact, which is a weapon that will help you to get through the difficulties, and furthermore, you can solve puzzles to get the treasure that will make you perfect the game called “Treasure Land”.

Propositions

Bridge(x): A bridge exists at the location x. If we want to find the clue and there is a river, we can pass the river

River(x): A river exists at location x. A river can block the way to find the clue.

Puzzle(x): A puzzle exists at location x. We can find clues to find the puzzle, if we solve the puzzle we can find the treasure.

Hazard(x): A hazard exists at location x. If the player gets a hazard, the player can not find the treasure.

Artifact(x): An artifact exists at location x. An artifact can help the player avoid the hazard.

Clue(x,y): A clue exists at location x pointing to location y. As long as the player finds the clue, the player can find the puzzle through the clue point x given at point y.

Treasure(x): Treasure at location x. The winning state.

Rainy_Wether(x) = True is the weather is rainy when the game is continuing.

Marsh(x) = True if the weather is rainy.

Constraints:

1) When there is a river and the player as location x does not have a bridge to pass the river, the player won't have access to the clue.

$$\text{River} \wedge \neg\text{Bridge} \rightarrow \neg\text{Clue}$$

2) If there is a river and we have a bridge, the player can have access to the clue.

$$\text{Bridge} \wedge \text{River} \rightarrow \text{Clue}$$

3) When the player has found the clue, the clue can leds the player to puzzle

$$\text{Clue} \rightarrow \text{Puzzle}$$

4) If we have access to the puzzle from the clue and we get no hazard, then we can have the treasure.

$$\text{Puzzle} \wedge \neg\text{Hazard} \rightarrow \text{Treasure}$$

5) When a hazard exists and the player has no artifact, the player can not have access to the hazard.

$\text{Hazard} \wedge \neg\text{Artifact} \rightarrow \neg\text{Treasure}$

6) If we have artifacts then we will not get into trouble.

$\text{Artifact} \rightarrow \neg\text{Hazard}$

7) When there is rainy weather, there will be marsh in the treasure island.

$\text{Rainy\_Weather} \rightarrow \text{Marsh}$

8) If the marsh appears and there is no artifact,then the player will not be able to find the clue.

$\text{Marsh} \wedge \neg\text{Artifact} \rightarrow \neg\text{Clue}$

9) When the player does not have the bridge to pass the mersh, the player cannot have access to the clue.

$\text{Marsh} \wedge \neg\text{Bridge} \rightarrow \neg\text{Clue}$

First-Order Extension

Propositions:

Player1(i): First player who joined the game at location i.

Player2(i): Second Player who joined the game at location i

Weapon: Players can use weapons to eliminate enemies.

Barbarian: These guys will attack you when they meet you

Barbarian_Chief: can be found by barbarians. Players can know the location of Zephyr by talking to the Barbarin_Chief.

Zephyr: Appears randomly anywhere on the map and does not move. When the player meets him, you must link up with another player to challenge him. Successful challenges earn you a respawn chance.

Respawn: come back to life from where you died.

Constraints:

Conditions of constraints:

- state all of the propositions: Barbarian, Barbarian_Chief, Player1, Player2, Zephyr.
- ∀x means everything included in the Treasure Island. Every propositions can be included in the set of x.

1) The Barbarians cannot be defeated if either player1 and player2 has no weapon.

$$\forall x(((\text{Player1}(x) \lor \text{Player2}(x)) \lor \neg\text{Weapon}(x)) \rightarrow \text{Barbarian}(x))$$

2) If either player1 and player2 alks to the barbarian chief, they can know the location of Zephyr.

$$\forall x(\text{Barbarian\_Chief}(x) \land (\text{Player1}(x) \lor \text{Player2}(x)) \rightarrow \text{Zephyr}(x))$$

3) If either Player1 or Player2 were missed, we can not defeat Zephyr, which will lead to not having a respawn chance.

$\forall y(Zephyr(x) \wedge (\neg Player(x) \vee \neg Player2(x)) \rightarrow \neg Respawn)$

4) The Barbarian can be defeated if either player1 or player2 brings the weapon.

$\forall x(Barbarian(x) \wedge Weapon \wedge (Player1(x) \vee Player2(x)) \rightarrow \neg Barbarian(x)$

5) Both players need to work together to defeat Zephyr so that they can earn a chance to respawn

$\forall x(Zephyr(x) \wedge (Player1(x) \wedge Player2(x)) \rightarrow Respawn)$

Model Exploration

1. Project Summary

This project creates a dynamic game setup where each "location" is populated with a set of game elements, and logical constraints determine valid configurations. The system:

• Defines locations and their elements programmatically.

• Establishes relationships and constraints between elements.

• Evaluates setups for validity and explores all possible configurations.

The project supports user interaction for customizing setups, testing specific conditions, and finding solutions programmatically.

2. Model Design

2.1 Elements

The model revolves around **game elements** representing features at each location. These elements include:

• **Physical Structures**: bridge, river

• **Challenges**: puzzle, hazard

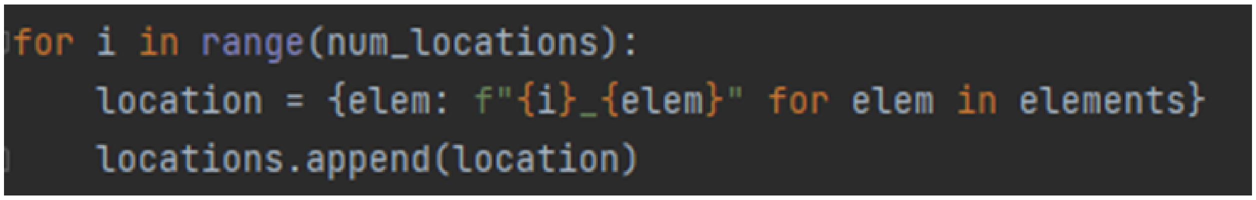
• **Rewards**: artifact, clue, treasure

• **Environmental Conditions**: rainy_weather, marsh

Each element is treated as a **binary variable** (True or False).

2.2 Locations

Each location is represented by a dictionary that maps element names to unique identifiers:



```
for i in range(num_locations):
    location = {elem: f"{i}_{elem}" for elem in elements}
    locations.append(location)
```

This ensures that every location's elements are unique across the game. For example:

- Location 0: { "bridge": "0_bridge", "river": "0_river", ... }
- Location 1: { "bridge": "1_bridge", "river": "1_river", ... }

The number of locations (num_locations) can be adjusted, dynamically scaling the game's complexity.

3. Constraints

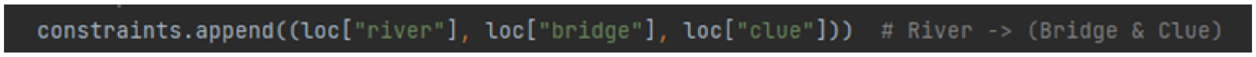
Constraints define logical relationships between elements within a single location or across locations. These rules shape the game's logic and valid setups.

3.1 Core Constraints

Constraints are expressed as logical implications:

1. River leads to bridge and clue:

$$\textbf{River} \rightarrow (\textbf{bridge} \wedge \textbf{clue})$$



```
constraints.append((loc["river"], loc["bridge"], loc["clue"]))  # River -> (Bridge & Clue)
```

2. Hazard negates artifact or treasure:

$$\textbf{hazard} \rightarrow \neg(\textbf{artifact} \vee \textbf{treasure})$$

This means if hazard is True, then neither artifact nor treasure can be True.

3. Rainy weather implies marsh:

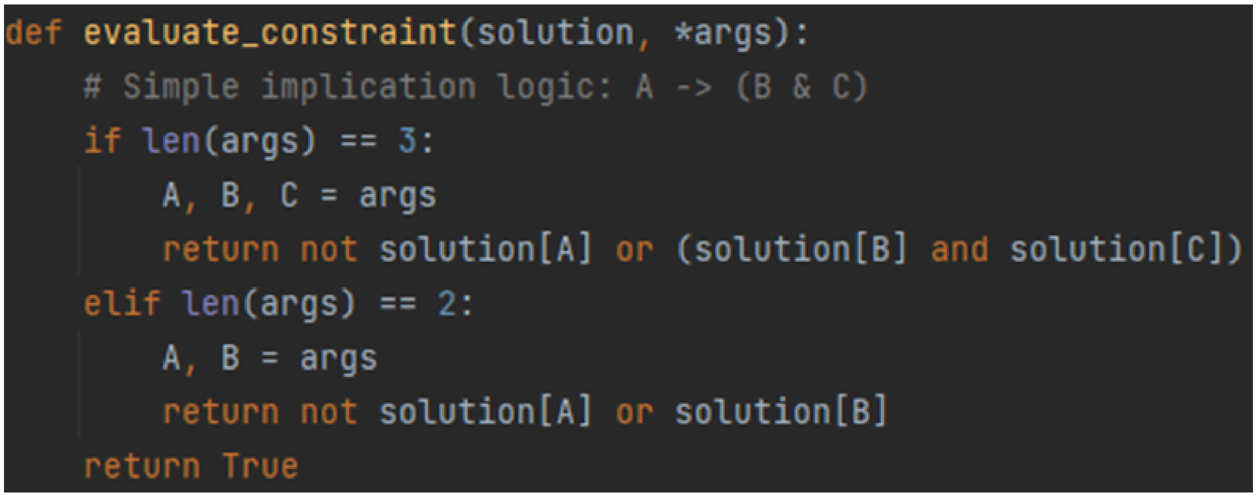
\text{rainy_weather} \rightarrow \text{marsh}

3.2 Logical Representation

The constraints are evaluated in Python using implication rules:

- **Implication ($A \rightarrow B$)**: not A or B
- **Conjunction ($A \wedge B$)**: A and B
- **Disjunction ($A \vee B$)**: A or B

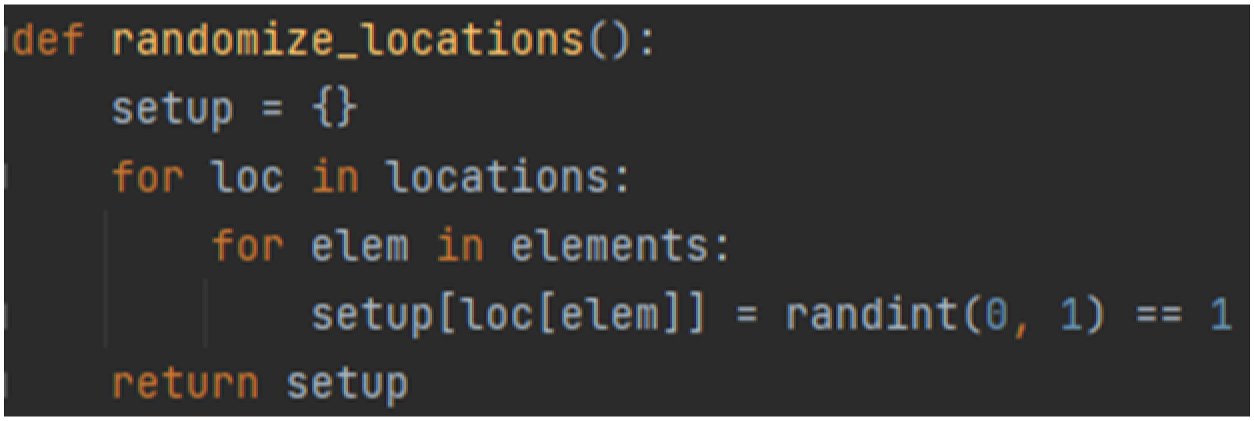
Example evaluation:


```python
def evaluate_constraint(solution, *args):
    # Simple implication logic: A -> (B & C)
    if len(args) == 3:
        A, B, C = args
        return not solution[A] or (solution[B] and solution[C])
    elif len(args) == 2:
        A, B = args
        return not solution[A] or solution[B]
    return True
```

4. System Functions

4.1 Random Setup

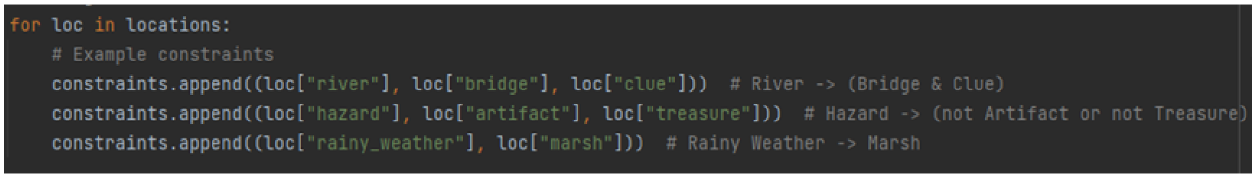
The function randomize_locations generates a random game state:


```python
def randomize_locations():
    setup = {}
    for loc in locations:
        for elem in elements:
            setup[loc[elem]] = randint(0, 1) == 1
    return setup
```

Each element in each location is assigned a random True/False value. This creates varied setups for testing.

4.2 Generating Constraints

The generate_constraints function creates all constraints for the current setup. It loops through each location:

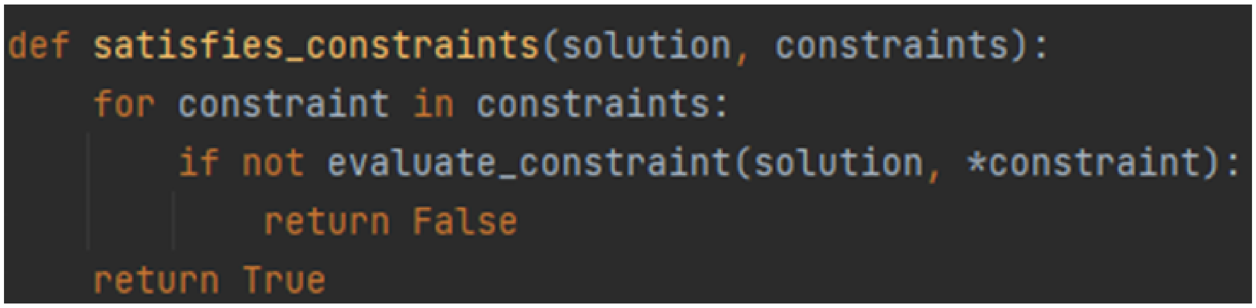


```python
for loc in locations:
    # Example constraints
    constraints.append((loc["river"], loc["bridge"], loc["clue"]))  # River -> (Bridge & Clue)
    constraints.append((loc["hazard"], loc["artifact"], loc["treasure"]))  # Hazard -> (not Artifact or not Treasure)
    constraints.append((loc["rainy_weather"], loc["marsh"]))  # Rainy Weather -> Marsh
```

This ensures all locations share the same logical rules.

4.3 Checking Constraints

To validate a setup, the satisfies_constraints function ensures all constraints hold:

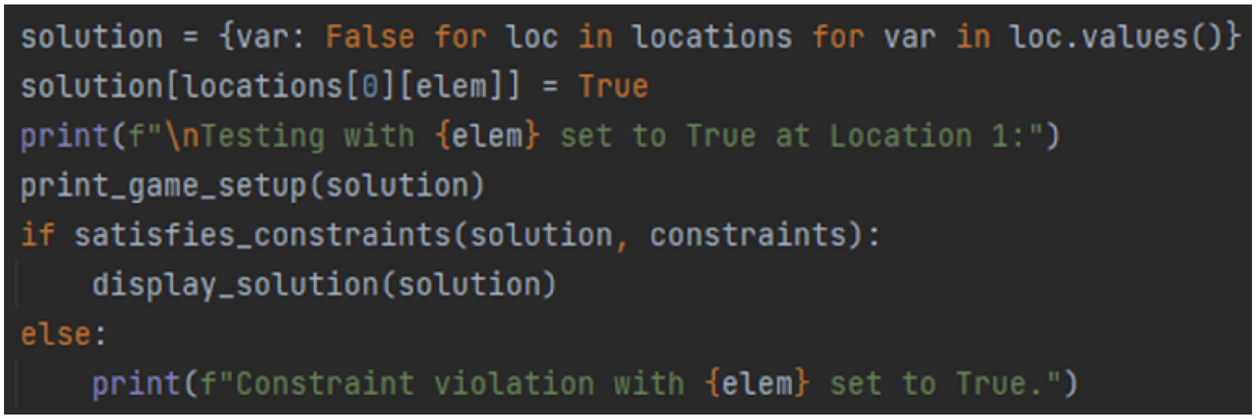


```python
def satisfies_constraints(solution, constraints):
    for constraint in constraints:
        if not evaluate_constraint(solution, *constraint):
            return False
    return True
```

5. Model Exploration

5.1 Testing Individual Elements

The function test_all_combos examines the effect of enabling each element in the first location:

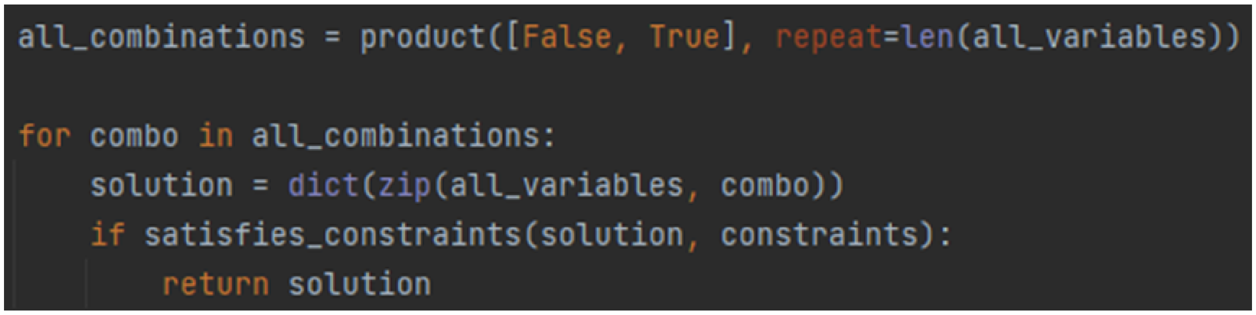


```python
solution = {var: False for loc in locations for var in loc.values()}
solution[locations[0][elem]] = True
print(f"\nTesting with {elem} set to True at Location 1:")
print_game_setup(solution)
if satisfies_constraints(solution, constraints):
    display_solution(solution)
else:
    print(f"Constraint violation with {elem} set to True.")
```

This tests how enabling a specific element impacts the constraints.

5.2 Full Solution Search

The function solve_game generates all possible combinations of True/False for every element:



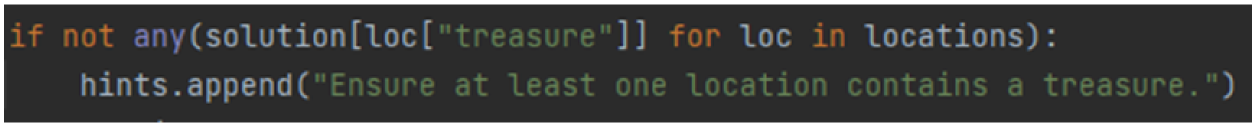
```python
all_combinations = product([False, True], repeat=len(all_variables))

for combo in all_combinations:
    solution = dict(zip(all_variables, combo))
    if satisfies_constraints(solution, constraints):
        return solution
```

This brute-force approach guarantees finding a valid solution, if one exists.

6. Victory Conditions

6.1 Checking Goals

The check_victory_conditions function ensures key objectives (e.g., finding treasure) are met:



```
if not any(solution[loc["treasure"]] for loc in locations):
    hints.append("Ensure at least one location contains a treasure.")
```

6.2 Displaying Results

The display_solution function presents the game state and victory status:

1. Victory Achieved:

• Treasure is present in at least one location.

• Displays the active elements at each location.

2. Hints:

• If goals are unmet, hints are provided for improvement.

Example Output:

```
Location 1: treasure, clue

Location 2: bridge, river

Hints:

  - Ensure at least one location contains a treasure.
```

7. Testing and Validation

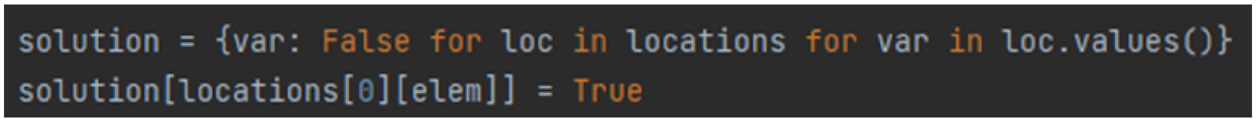
7.1 Constraints Validation

Testing ensures constraints behave as expected:

• Enable/disable specific elements.

• Validate outcomes against constraints.

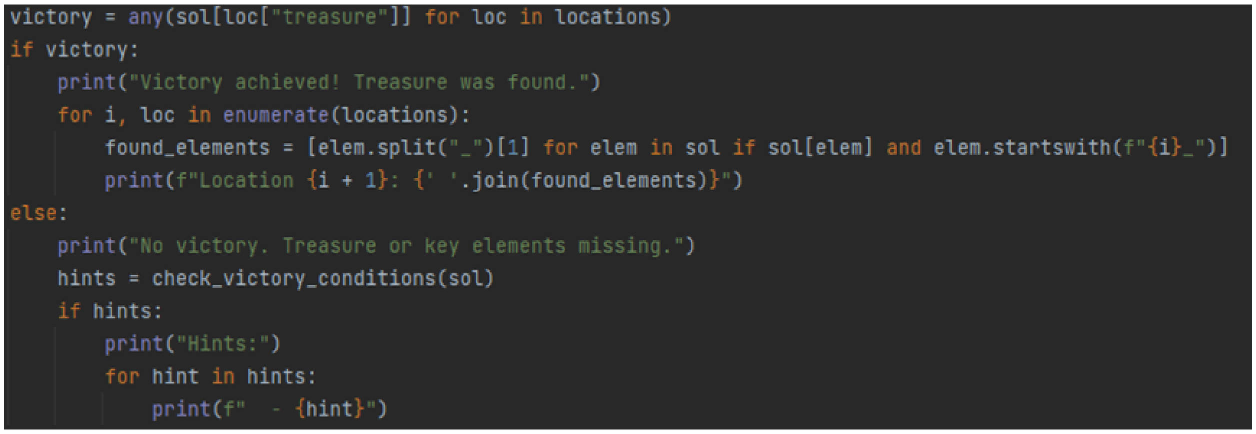
Example:



```
solution = {var: False for loc in locations for var in loc.values()}
solution[locations[0][elem]] = True
```

7.2 Victory Testing

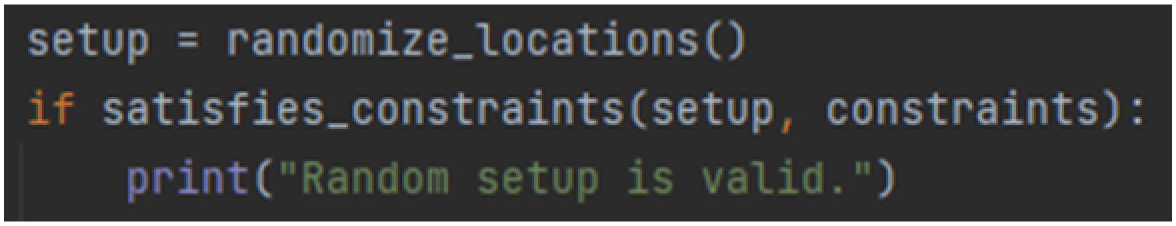
Ensures treasure-related goals are met:



```
victory = any(sol[loc["treasure"]] for loc in locations)
if victory:
    print("Victory achieved! Treasure was found.")
    for i, loc in enumerate(locations):
        found_elements = [elem.split("_")[1] for elem in sol if sol[elem] and elem.startswith(f"{i}_")]
        print(f"Location {i + 1}: {' '.join(found_elements)}")
else:
    print("No victory. Treasure or key elements missing.")
    hints = check_victory_conditions(sol)
    if hints:
        print("Hints:")
        for hint in hints:
            print(f"  - {hint}")
```

7.3 Randomized Testing

Simulates random game states and checks validity:

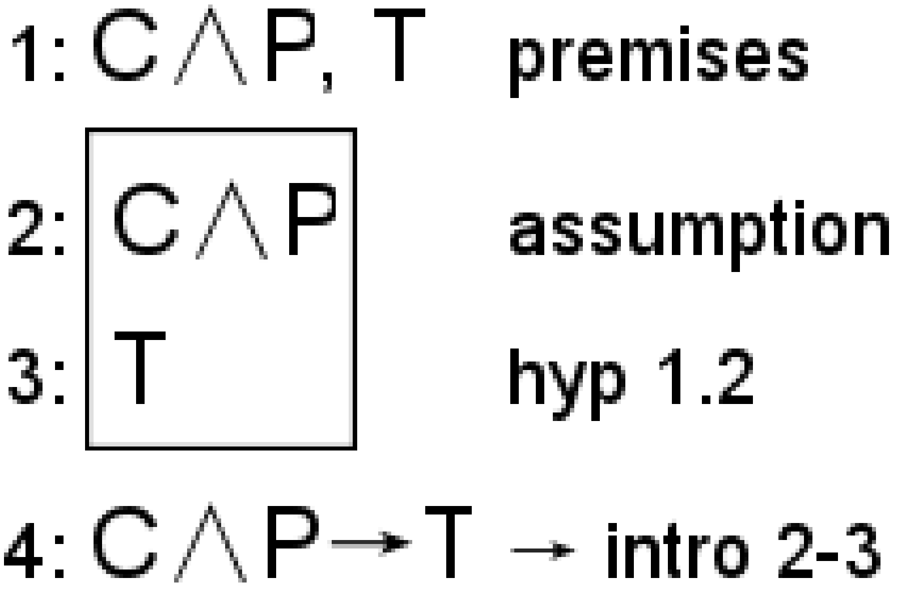


```
setup = randomize_locations()
if satisfies_constraints(setup, constraints):
    print("Random setup is valid.")
```

Jape Proof Ideas

1) This proof is about finding the treasure, which needs two elements, clue and puzzle. If the player has the clue, which is able to lead the player to find where the puzzle is, and have the puzzle solved, these two conditions can help the player find the treasure.

C = Clue, T = Treasure, P = puzzle



1: $C \wedge P, T$ premises

2: $C \wedge P$ assumption

3: T hyp 1.2

4: $C \wedge P \rightarrow T$ $\rightarrow$ intro 2-3

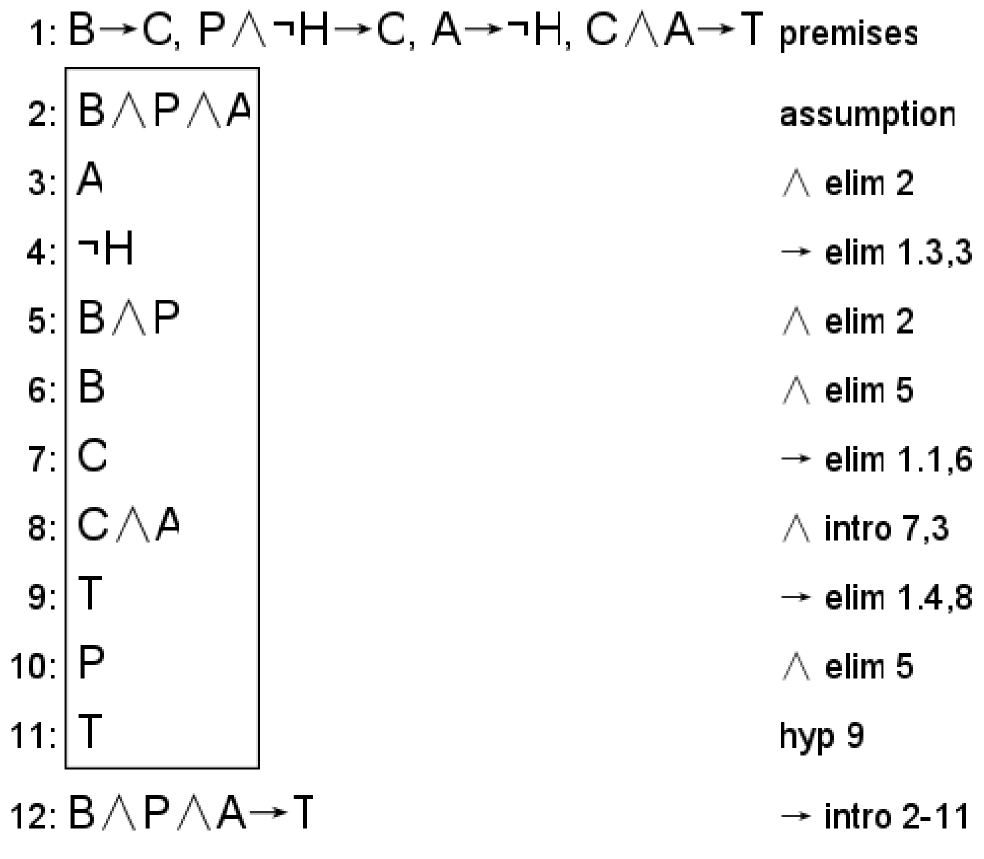
2) Since we are talking about finding the treasure, This proof is meant to be a contingency encountered while searching for the treasure. In this scenario, the contingency is having a hazard, which is a possible to kill the player. In order to avoid hazards, the player needs to use artifact. Therefore, the idea is that if we have an artifact, we can avoid hazard. If we have no hazard and we have solved the puzzle(True) then we can get the treasure. So we can conclude that if we have an artifact to avoid hazard from hurting the player and solve the puzzle(The puzzle is true), then we can get the treasure.

A = artifact, H = hazard, P = puzzle, T = treasure

1:	$A \rightarrow \neg H,\ P \wedge \neg H \rightarrow T$	premises
2:	$A \wedge P$	assumption
3:	P	$\wedge$ elim 2
4:	A	$\wedge$ elim 2
5:	$\neg H$	$\rightarrow$ elim 1.1,4
6:	$P \wedge \neg H$	$\wedge$ intro 3,5
7:	T	$\rightarrow$ elim 1.2,6
8:	$A \wedge P \rightarrow T$	$\rightarrow$ intro 2-7

3) The idea of the proof is that If there is a river, we need a bridge to cross the river to find the clue, so we can find a bridge where there is a river in order to find the clue. In addition, when we find the puzzle and we solve it meanwhile if we get no hazard, we can find the clue. The third premise is that we can find the artifact to avoid hazard. The last one is after we get the clue and artifact, we can find the treasure at the end. Therefore, we can get the conclusion that if we have a bridge and an artifact with puzzle solved, we can find the treasure.

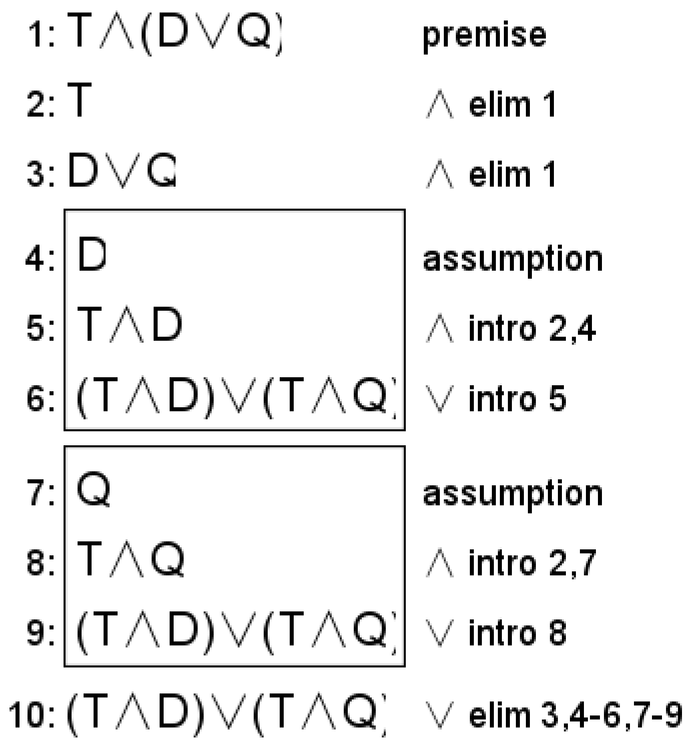
B = bridge, C = clue, P = puzzle, H = hazard, A = artifact, T = treasure



1:	$B \to C,\ P \land \neg H \to C,\ A \to \neg H,\ C \land A \to T$	premises
2:	$B \land P \land A$	assumption
3:	A	$\land$ elim 2
4:	$\neg H$	$\to$ elim 1.3,3
5:	$B \land P$	$\land$ elim 2
6:	B	$\land$ elim 5
7:	C	$\to$ elim 1.1,6
8:	$C \land A$	$\land$ intro 7,3
9:	T	$\to$ elim 1.4,8
10:	P	$\land$ elim 5
11:	T	hyp 9
12:	$B \land P \land A \to T$	$\to$ intro 2-11

4) The idea of this proof: Treasure(T), the object of the game, holds and either the Resurrection stones(D) or Marsh(Q) can alter the process of the game, where Resurrection stones can give the player a chance to reborn and Marsh could possibly kill the player which also means end the game, then the game process can be split into two scenarios where Treasure(T) will occur along with Resurrection stones(D) and Marsh(Q).

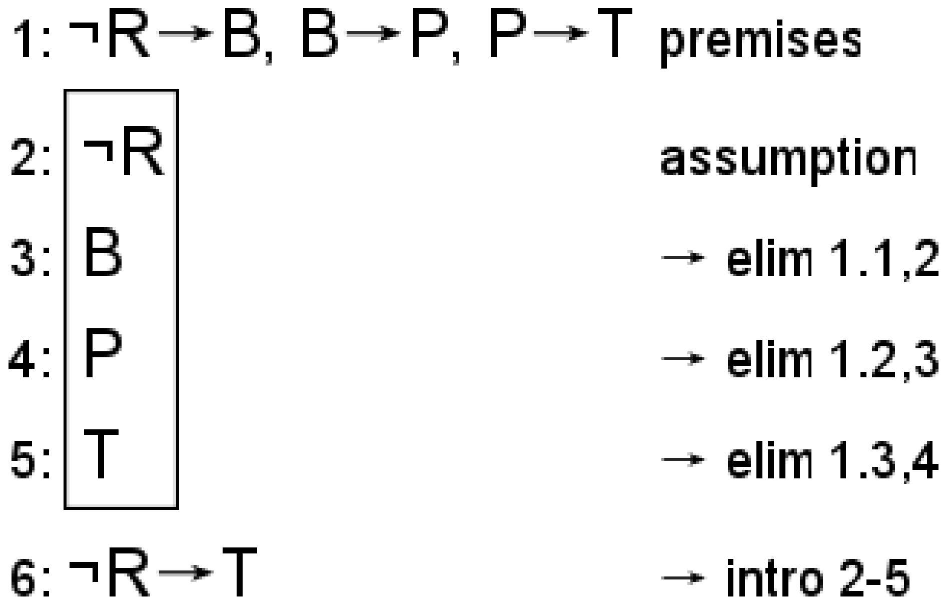
T = Treasure; D =Resurrection stones; Q = Marsh



1:	$T \wedge (D \vee Q)$	premise
2:	T	$\wedge$ elim 1
3:	$D \vee Q$	$\wedge$ elim 1
4:	D	assumption
5:	$T \wedge D$	$\wedge$ intro 2,4
6:	$(T \wedge D) \vee (T \wedge Q)$	$\vee$ intro 5
7:	Q	assumption
8:	$T \wedge Q$	$\wedge$ intro 2,7
9:	$(T \wedge D) \vee (T \wedge Q)$	$\vee$ intro 8
10:	$(T \wedge D) \vee (T \wedge Q)$	$\vee$ elim 3,4-6,7-9

5) The idea of the proof: If we have no river(¬R(x)), it means we have a bridge(B(x)), and a bridge can help us to find the puzzle. As long as we solve the puzzle(true), we can find the treasure. The goal is to prove that if there is no river, we can find the treasure.

R = river; B = bridge; P = puzzle; T = treasure



1:	$\neg R \rightarrow B,\ B \rightarrow P,\ P \rightarrow T$	premises
2:	$\neg R$	assumption
3:	B	$\rightarrow$ elim 1.1,2
4:	P	$\rightarrow$ elim 1.2,3
5:	T	$\rightarrow$ elim 1.3,4
6:	$\neg R \rightarrow T$	$\rightarrow$ intro 2-5