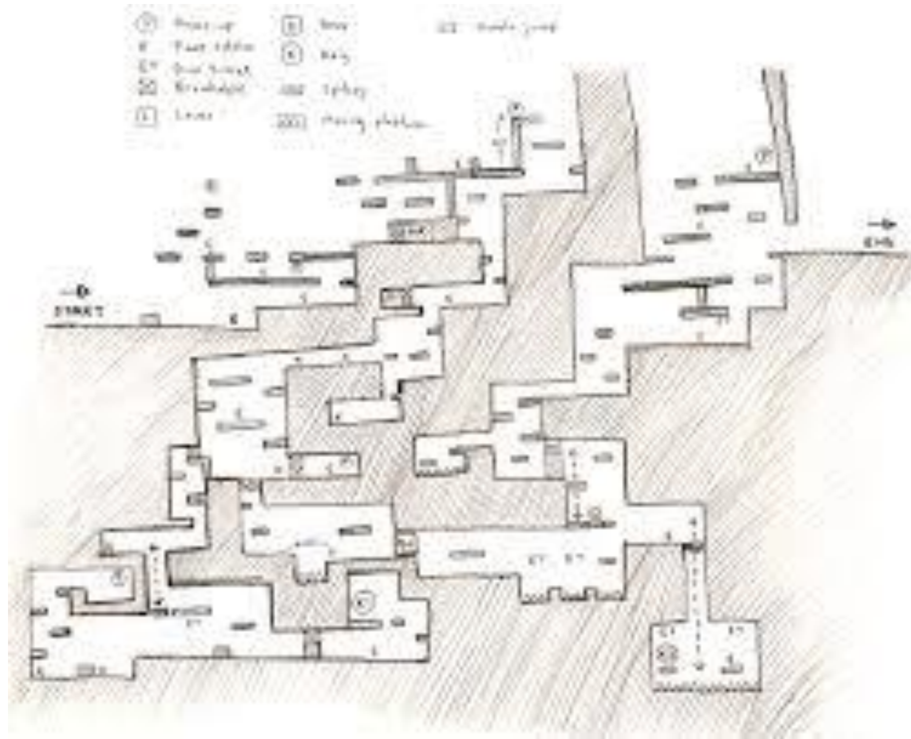
Using Model Counting for Game Development: Quantifying Difficulty of 2D Platformer Levels for Diverse Playable Characters

Aditya Patil (MIT Vishwashanti Gurukul,India)
Mark Santolucito (Barnard College,USA)



Problem Statement

- Game designers in the past have struggled to strike a balance in generating levels for diverse playable characters.

Hypothesis

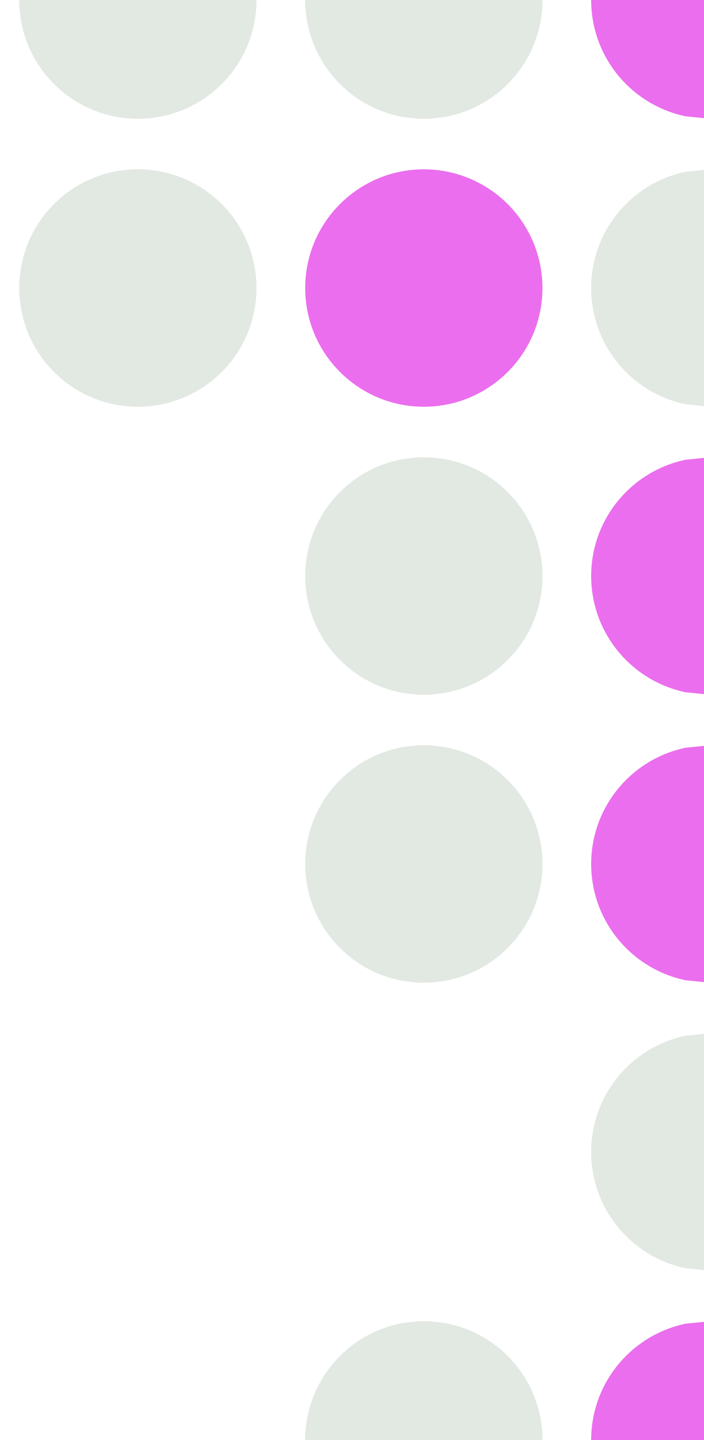
- Using model counting, evaluation of different levels will become easier for diverse playable characters.

Super Mario Level Balance

- Dynamic Difficulty Adjustments: The levels adapt to player performance offering extra power-ups or minimizing challenges when players struggle.
- Player Experience: Gameplay elements like enemy behavior or available items adjust based on how well or poorly the player is performing.
- Balanced Skill Levels: Ensures that both beginners and advanced players can enjoy the game by modifying difficulty to suit the player's needs.

Proof of Concept

- Before we started with evaluating a 2D platformer level difficulty we tested the model counting constraints with a simple game of tic-tac-toe



Tic Tac Toe Test

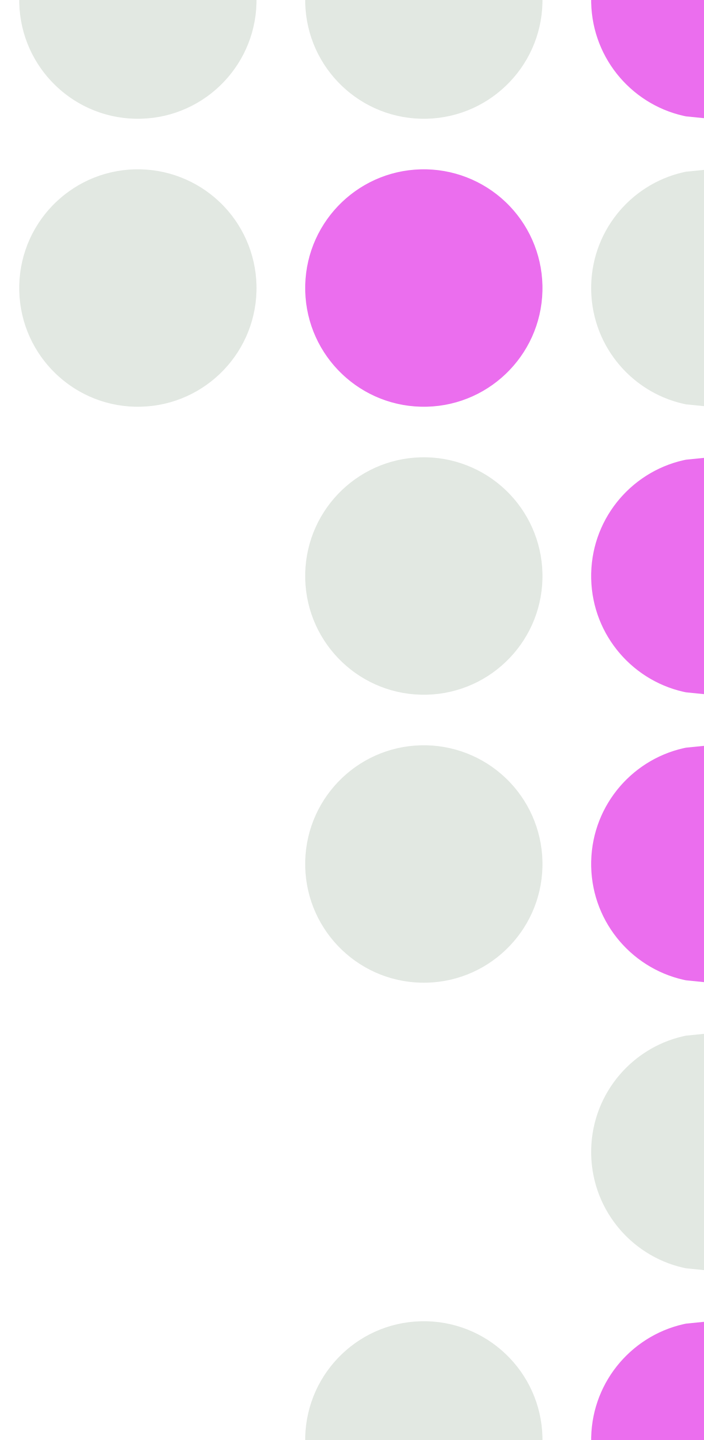
During this test I set up 3 different constraints:

- Cell Occupancy: Each cell must contain either an "X" or an "O"but not both.
- Turn Alternation: The number of "X's" and "O's" on the board must differ by at most 1 to ensure correct turn alternation.
- Single Winner: Only one player can win, either "X" or "O", but not both simultaneously.

```
from z3 import *

X = [[Bool(f"X_{i}_{j}") for j in range(3)] for i in range(3)]
O = [[Bool(f"O_{i}_{j}") for j in range(3)] for i in range(3)]

solver = Solver()

for i in range(3):
    for j in range(3):
        solver.add(Or(X[i][j], O[i][j]))
        solver.add(Not(And(X[i][j], O[i][j])))

solver.add(Sum([If(X[i][j], 1, 0) for i in range(3) for j in range(3)]) -
           Sum([If(O[i][j], 1, 0) for i in range(3) for j in range(3)]) <= 1)
solver.add(Sum([If(O[i][j], 1, 0) for i in range(3) for j in range(3)]) -
           Sum([If(X[i][j], 1, 0) for i in range(3) for j in range(3)]) <= 0)

def is_winner(player):
    row_wins = [And(player[i][0], player[i][1], player[i][2]) for i in range(3)]
    col_wins = [And(player[0][j], player[1][j], player[2][j]) for j in range(3)]
    diag1_win = And(player[0][0], player[1][1], player[2][2])
    diag2_win = And(player[0][2], player[1][1], player[2][0])
    return Or(*row_wins, *col_wins, diag1_win, diag2_win)

X_wins = is_winner(X)
O_wins = is_winner(O)

solver.add(Not(And(X_wins, O_wins)))

def avoid_solution(model):
    constraints = []
    for i in range(3):
        for j in range(3):
            x_val = model.evaluate(X[i][j])
            o_val = model.evaluate(O[i][j])
            constraints.append(Or(X[i][j] != x_val, O[i][j] != o_val))
    return Or(constraints)

def count_solutions():
  counter = 0
  while solver.check() == sat:
      counter += 1
      model = solver.model()
      board = [["X" if model.evaluate(X[i][j]) else "O" if model.evaluate(O[i][j]) else "-" for j in

      solver.add(avoid_solution(model))
  return counter

solver.push()
totalPlays = count_solutions()
print(f"Total number of valid configurations: {totalPlays}")

solver.pop()
solver.push()
solver.add(O_wins)
oWins = count_solutions()
print(f"Total number of O win options: {oWins}")

solver.pop()
solver.add(X_wins)
xWins = count_solutions()
print(f"Total number of X win options: {xWins}")

print(f"difficulty for O: {1 - (oWins+0.0)/totalPlays}")
print(f"difficulty for X: {1 - (xWins+0.0)/totalPlays}")
```

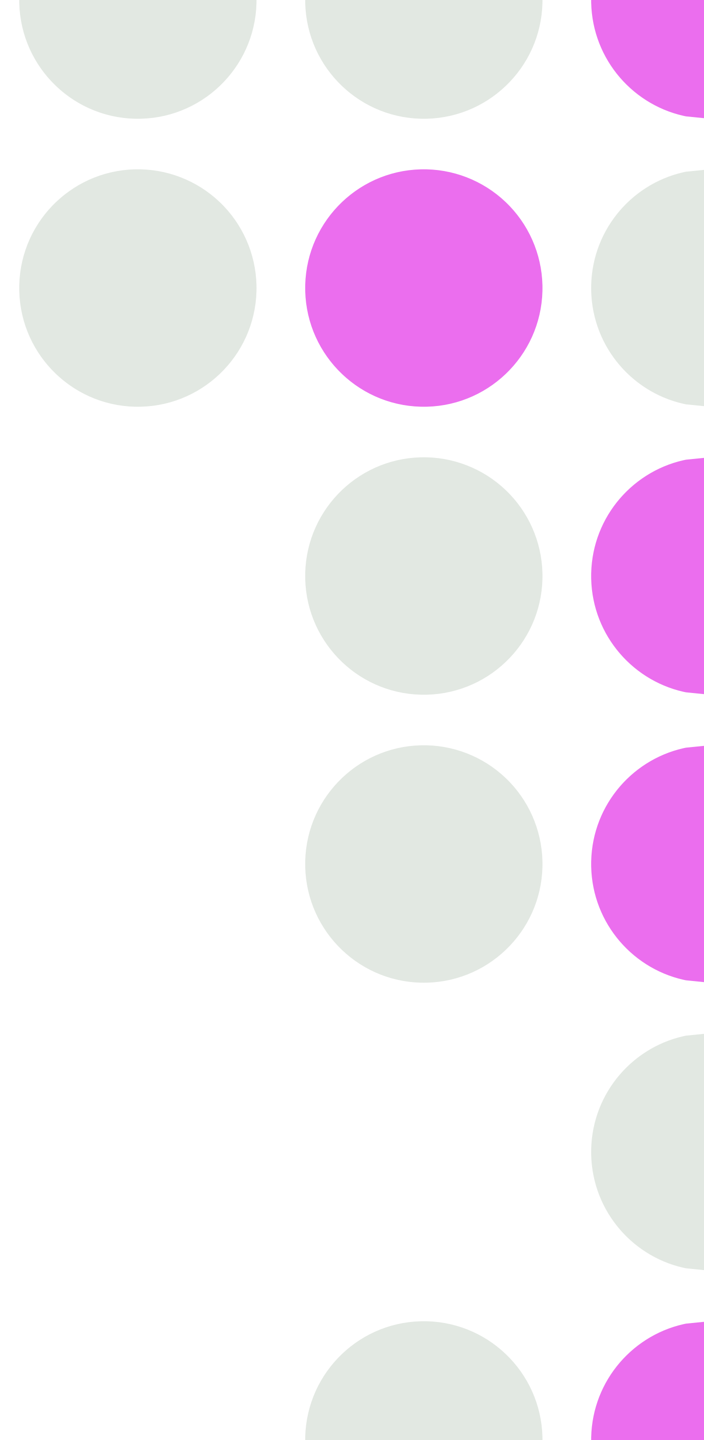
Why Satisfiability modulo theories (SMT) Solvers Are Perfect for Assessing Game Level Difficulty

- SMT solvers are great for figuring out game level difficulty because they can handle many complex rules and balance different factors effectively. They help you model different aspects of the game level, making it easier to see how hard the level is.

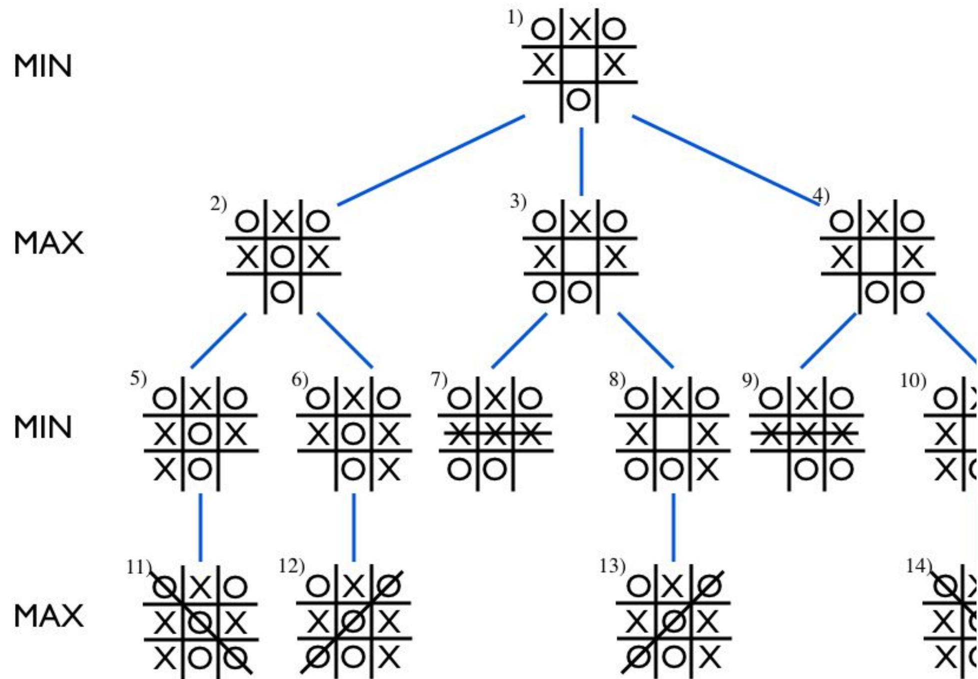


Why Z3 was chosen?

- Z3 is an efficient SMT solver
- The usage of Z3 with Python allows for quick prototyping and easy demonstration of formal methods in tic-tac-toe
- Z3 was chosen over other alternatives was due to its strong community support for a first time explorer like me along with limited time preventing a thorough evaluation of other solvers.



In-Depth



- Tic tac toe is a solved game and the optimal strategy is already known
- Z3 isn't the best tool for analyzing tic-tac-toe but it does us a proof of concept
- Our goal with Z3 is to later extend this approach to a 2D platformer

Constraint #0 Setting Up the Grid

- This code creates two 3x3 matrices, X and O, where each element is a Boolean variable representing whether player X or player O occupies a specific cell on the Tic-Tac-Toe board.
- The solver then adds constraints ensuring that each cell on the board is either occupied by X or O (but not both), enforcing that no cell can be occupied by both players simultaneously.

```
X = [[Bool(f"X_{i}_{j}") for j in range(3)] for i in range(3)]
O = [[Bool(f"O_{i}_{j}") for j in range(3)] for i in range(3)]

solver = Solver()

for i in range(3):
    for j in range(3):
        solver.add(Or(X[i][j], O[i][j]))
        solver.add(Not(And(X[i][j], O[i][j])))
```

Constraint #1: Defining Who Starts First

- These lines add constraints to ensure that the number of X moves is either equal to or one more than the number of O moves, enforcing the rule that players take turns and X always goes first.

```
solver.add(Sum([If(X[i][j], 1, 0) for i in range(3) for j in range(3)]) -
           Sum([If(O[i][j], 1, 0) for i in range(3) for j in range(3)]) <= 1)
solver.add(Sum([If(O[i][j], 1, 0) for i in range(3) for j in range(3)]) -
           Sum([If(X[i][j], 1, 0) for i in range(3) for j in range(3)]) <= 0)
```

Constraint #2: Defining the Winning Condition

- This function "is_winner"` checks if a given player (X or O) has won the game by forming a complete row, column, or diagonal. It returns `True` if any of these win conditions are met.
- The variables "X_wins" and "O_wins" are then set to indicate whether player X or player O has won, respectively.

```
def is_winner(player):
    row_wins = [And(player[i][0], player[i][1], player[i][2]) for i in range(3)]
    col_wins = [And(player[0][j], player[1][j], player[2][j]) for j in range(3)]
    diag1_win = And(player[0][0], player[1][1], player[2][2])
    diag2_win = And(player[0][2], player[1][1], player[2][0])
    return Or(*row_wins, *col_wins, diag1_win, diag2_win)

X_wins = is_winner(X)
O_wins = is_winner(O)
```

Constraint #3: Avoiding Pre-Existing Solutions

- The avoid_solution function creates constraints to avoid a specific Tic-Tac-Toe board configuration that has already been found.
- It does this by comparing each cell's value in the current model (board) and generating a constraint that forces at least one cell to have a different value in the next solution, ensuring uniqueness.

```
def avoid_solution(model):
    constraints = []
    for i in range(3):
        for j in range(3):
            x_val = model.evaluate(X[i][j])
            o_val = model.evaluate(O[i][j])
            constraints.append(Or(X[i][j] != x_val, O[i][j] != o_val))
    return Or(constraints)
```

Exploring all Possible Solutions

- The "count_solutions" function counts the total number of valid Tic-Tac-Toe board configurations.
- It does this by repeatedly checking if the current constraints are satisfiable "sat", incrementing a counter each time a valid configuration is found. It then adds a constraint to avoid the current solution using "avoid_solution", ensuring all unique configurations are counted.

```
def count_solutions():
  counter = 0
  while solver.check() == sat:
      counter += 1
      model = solver.model()
      board = [["X" if model.evaluate(X[i][j]) else "O" if model.evaluate(O[i][j]) else "-" for j in range(3)] for i in range(3)]

      solver.add(avoid_solution(model))
  return counter
```

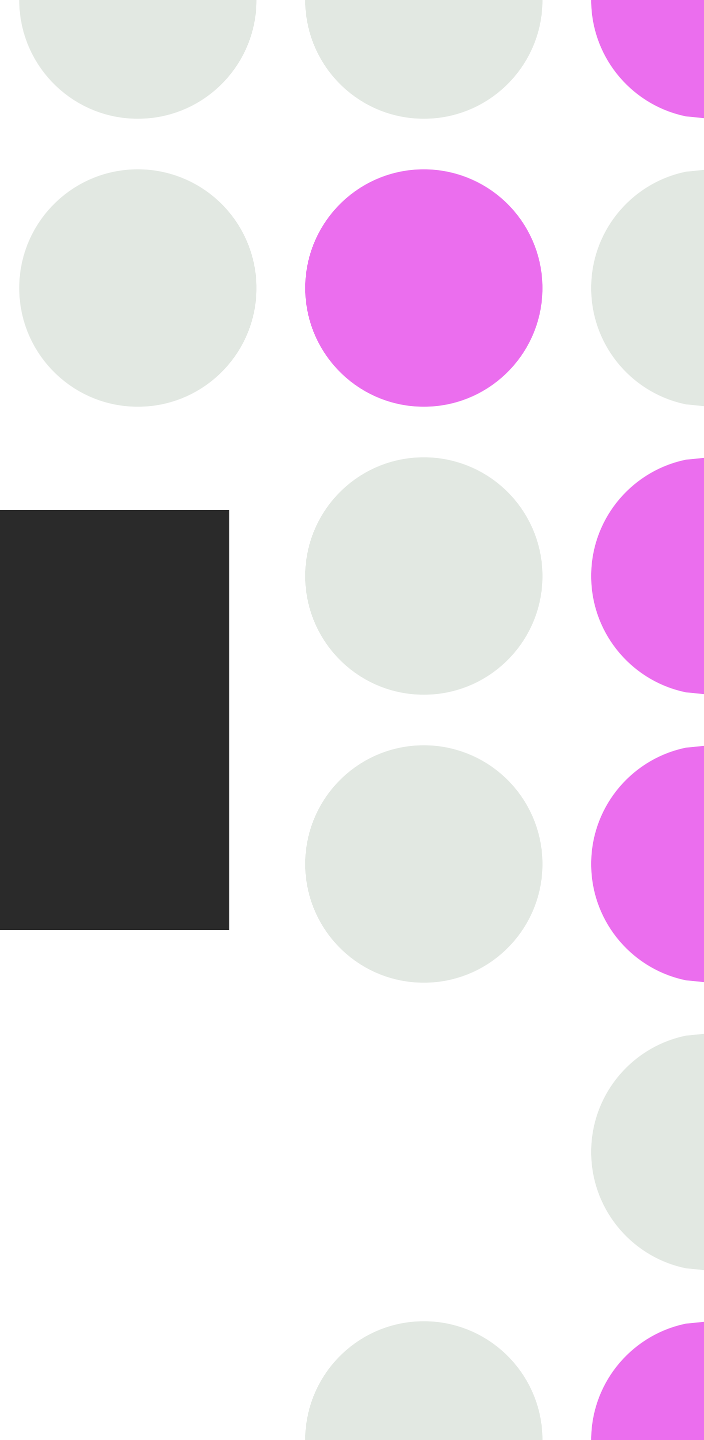
Assessing Difficulty Level

- `solver.push()` saves the current state of the solver's constraints. `
- totalPlays = count_solutions()` counts the total number of valid board configurations, and this number is printed.
- `solver.pop()` restores the solver to the previous state before any additional constraints were added.
- The process repeats twice, first with an added constraint for O winning (`solver.add(O_wins)`) to count `oWins`, and then with a constraint for X winning (`solver.add(X_wins)`) to count `xWins`
- Finally, the difficulty for each player is calculated by determining the proportion of winning configurations relative to the total, where a lower proportion of winning configurations indicates a higher difficulty. These difficulty levels are then printed.

```
solver.push()
totalPlays = count_solutions()
print(f"Total number of valid configurations: {totalPlays}")

solver.pop()
solver.push()
solver.add(O_wins)
oWins = count_solutions()
print(f"Total number of O win options: {oWins}")

solver.pop()
solver.add(X_wins)
xWins = count_solutions()
print(f"Total number of X win options: {xWins}")

print(f"difficulty for O: {1 - (oWins+0.0)/totalPlays}")
print(f"difficulty for X: {1 - (xWins+0.0)/totalPlays}")
```

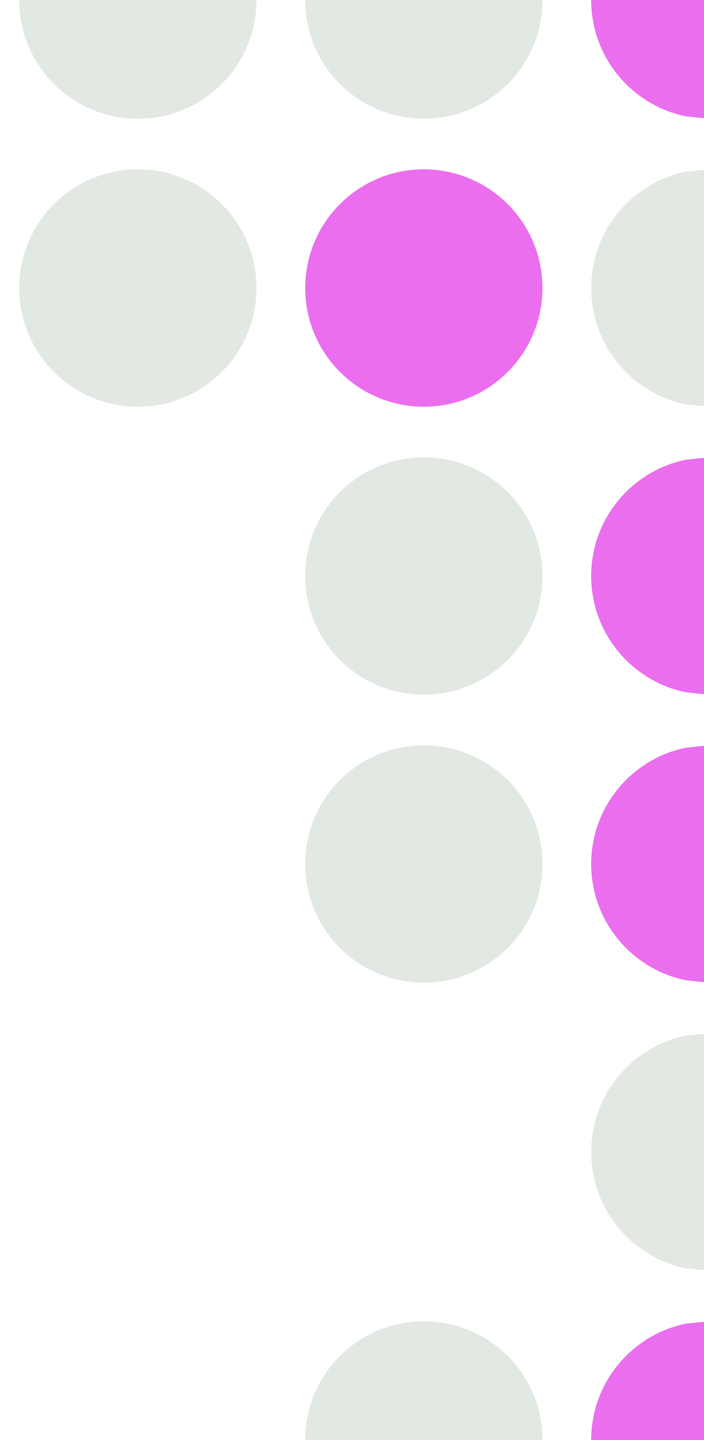
Test Results

```
Total number of valid configurations: 90
Total number of O win options: 12
Total number of X win options: 62
difficulty for O: 0.8666666666666667
difficulty for X: 0.3111111111111111
```



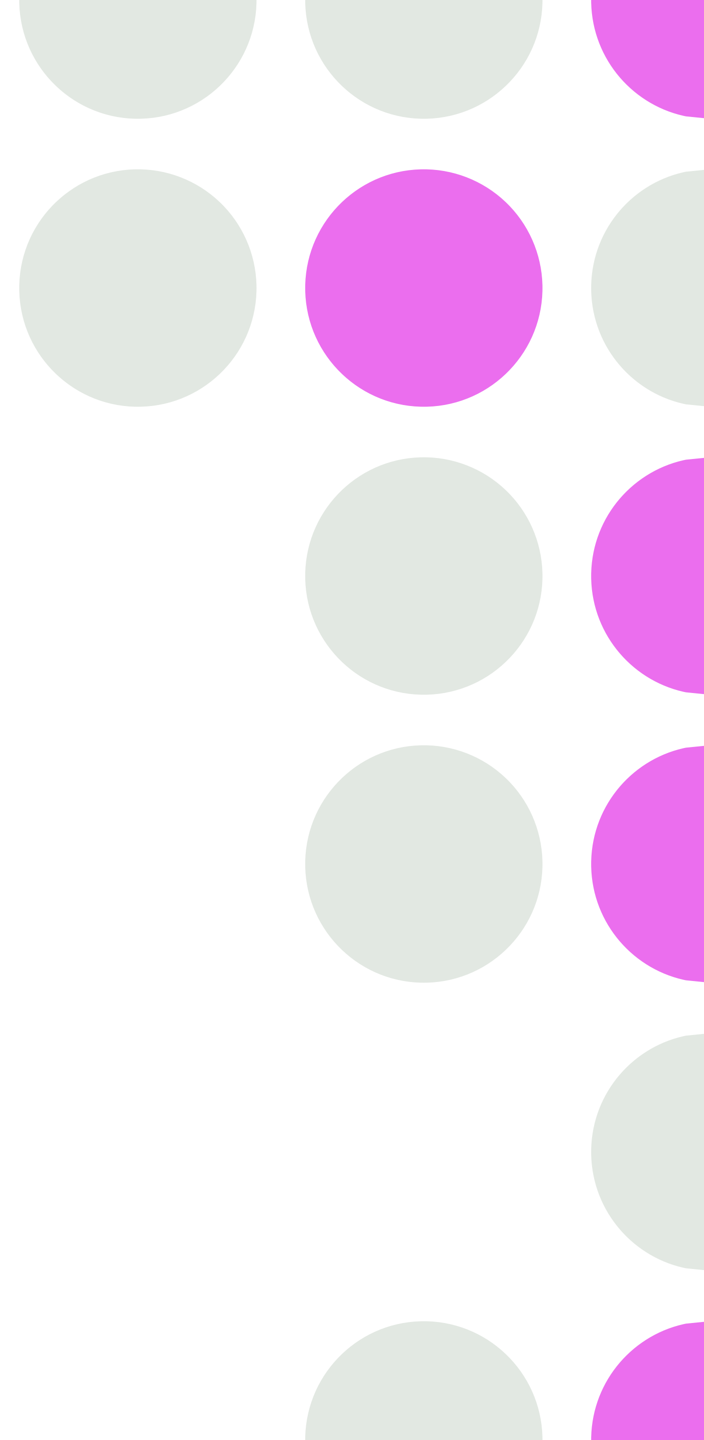
Conclusion from this Test

- The higher difficulty for O indicates that it is more challenging for O to win, which aligns with the first-move advantage generally seen in Tic-Tac-Toe.
- The Z3 Solver is highly effective for solving combinatorial problems like Tic-Tac-Toe, providing precise results for various configurations.
- This test proves that we can use model counting to assess game difficulty for different playable characters with certain constraints.



2D Platformer Test

- Keeping the previous test in mind, we did the same thing for a simple 2D platformer with all of the initial experiments done with setting up constraints.



Grid Setup

These lines of code:

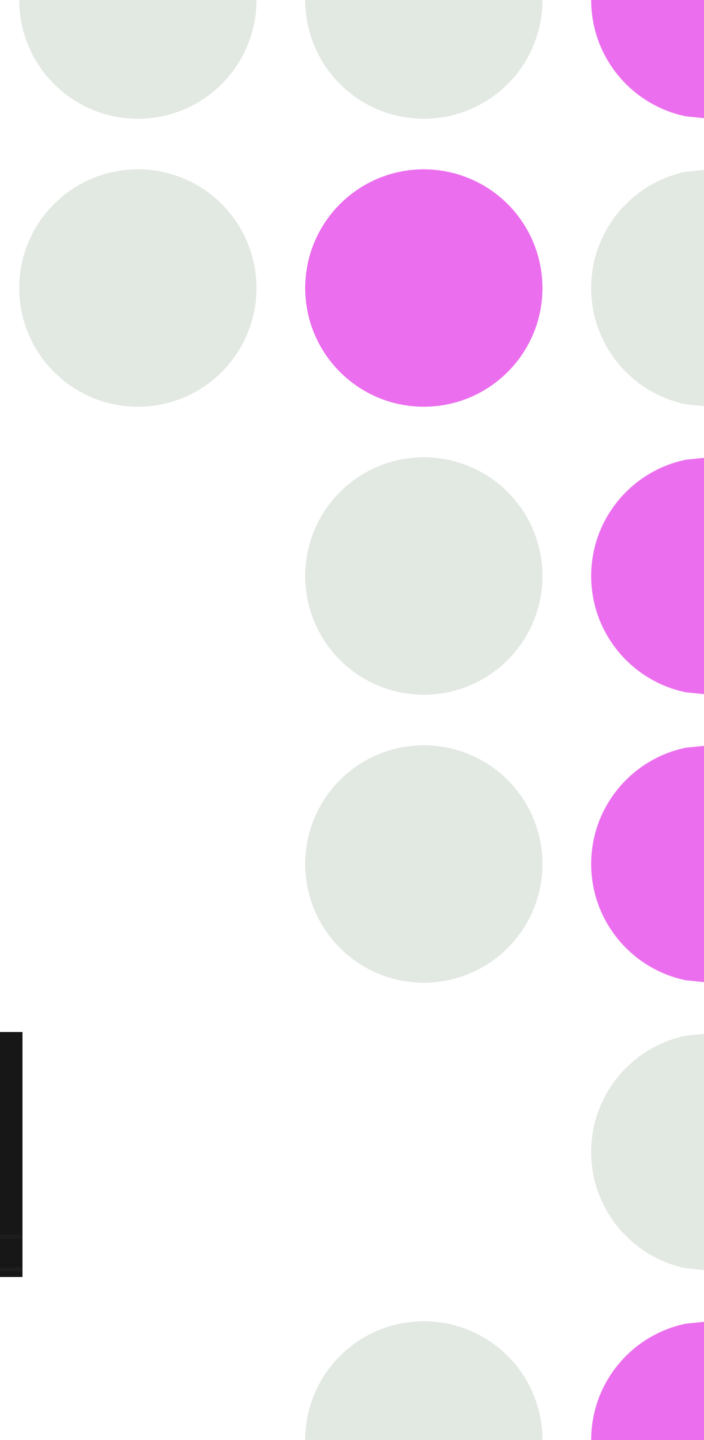
Defines the grid dimensions: 2 rows, 4 columns

Set number of moves to 5

Creates a 3D grid `Tank` with Boolean variables for each cell and time step

Initializes the Z3 solver

```
rows, cols = 2, 4
moves = 5

Tank = [[[Bool(f"Tank_{i}_{j}_{t}") for t in range(moves+1)] for j in range(cols)] for i in range(rows)]

solver = Solver()
```



Constraint #1 and #2

- Set initial Tank position: Ensure Tank starts at (1, 0) at time step 0.
- Unique position constraint: Ensure Tank is in only one cell per time step.

```
solver.add(Tank[1][0][0] == True)

for t in range(moves+1):
    for i in range(rows):
        for j in range(cols):
            other_positions = [Tank[x][y][t] for x in range(rows) for y in range(cols) if not (x == i and y == j)]
            solver.add(Implies(Tank[i][j][t], Not(Or(other_positions))))
```

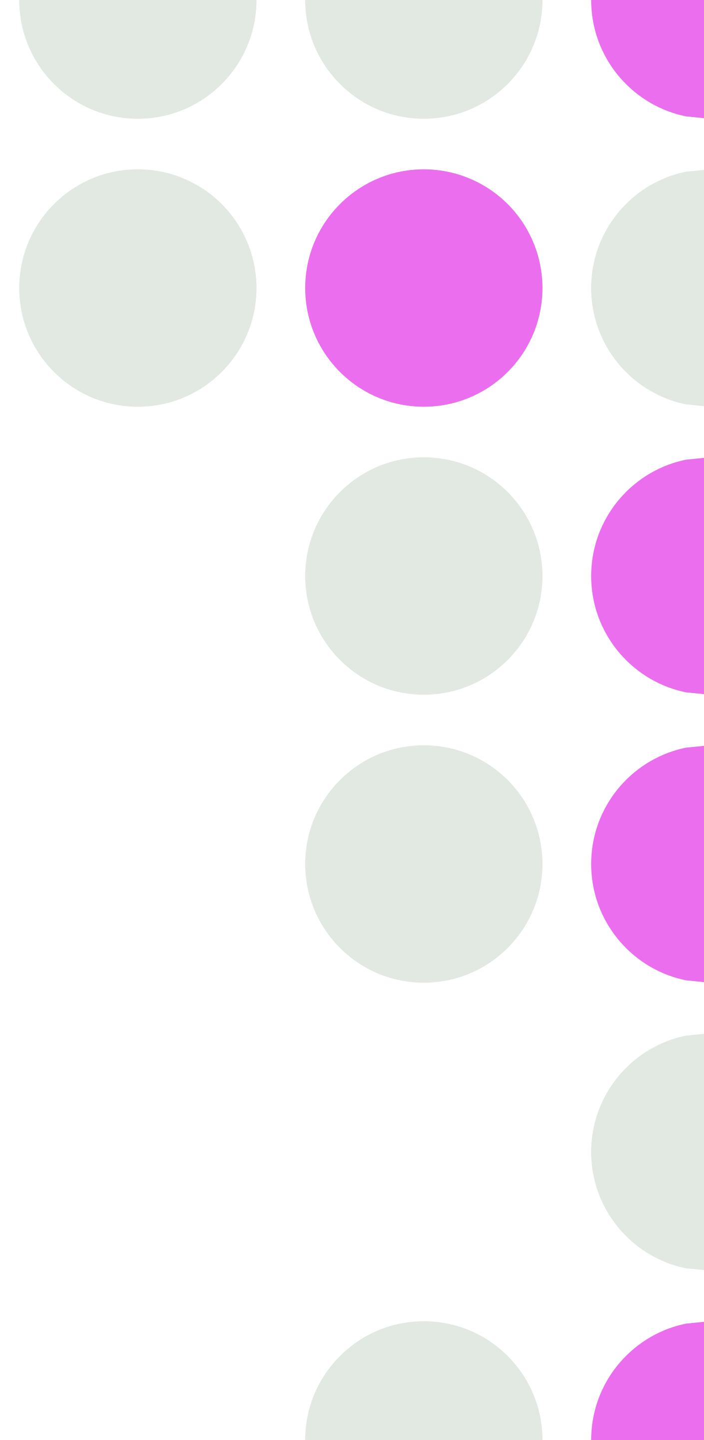
Movement Rules and Goal Position

- Movement Constraints: Ensure Tank moves right or stays in place.
- Goal Position: Ensure Tank reaches (1, 2) at final time step.

```
for t in range(1, moves+1):
    for j in range(1, cols):
        solver.add(Implies(Tank[1][j][t], Or(Tank[1][j-1][t-1], Tank[1][j][t-1])))

solver.add(Tank[1][2][moves] == True)
```

Counting Valid Paths for the Tank

- Count Valid Paths: The code than iteratively solves the constraints to find all valid paths for the Tank.
- Exclude Solutions: Adds a constraint to exclude the current solution from future checks.
- Return Path Count: Returns the total number of valid paths found and prints the number

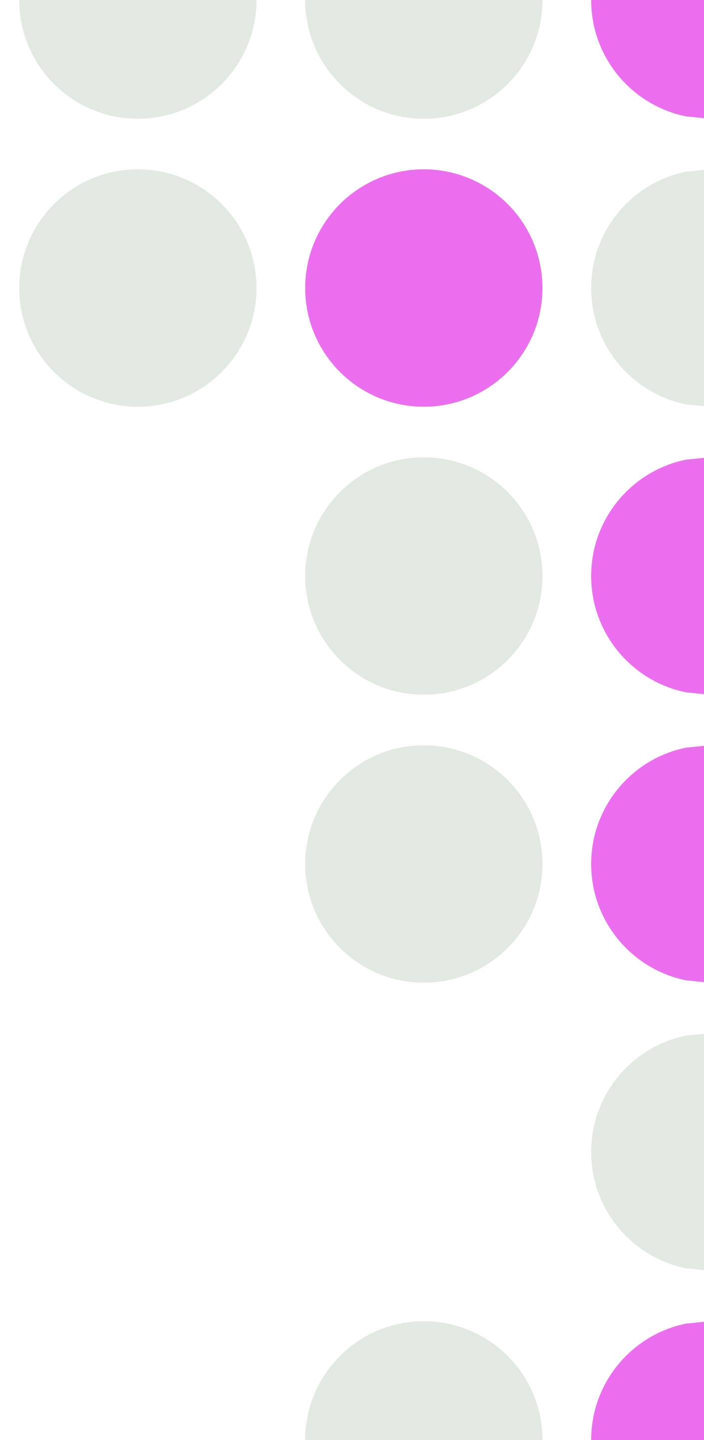


Test Result

```
Total number of valid paths for Tank: 55
```

Future Work

- The same method can be applied for finding the valid number of paths for an agile character
- Ways to optimize the model counting solution
- Using different factors to get level difficulty



THANK YOU