# Project Summary

This project aims to figure out how many ways you can win a Tetris game given a certain win condition for the player. Our definition of a “win” is to minimise height over a set number of rounds (this is temporary, I put this sentence in for completeness we can change the win condition later if we need). T-spins, which are last minute rotations, and sliding will not be calculated as potential moves. Our solution will take a given grid configuration, such as a board size of 4x6. The order in which the pieces will be placed will be randomized to best simulate an actual game of Tetris.

# Propositions

Tetris are placed on the grid.

* Blocked row, column – This is true if this grid location is already occupied by a block

When a row is filled it is cleared

* Rowclear row – This is true if a row has been filled

Game ends when Tetris pieces stack too high

* Fail column –This is true if the column has reached out of bounds (game over)

A Tetris piece cannot be placed in or through existing blocks on the grid.

* T p,I –This is true if the set p can be placed given the type of piece
* There are 7 Tetris pieces which all have 4 orientations: 0-18 each denoting a different piece # starting at 0 just like python
* Technically divided into a few sub-sections. E.g:0-3 denotes different orientations of T piece and 4,5 denotes orientations of I piece.

**Tetris Pieces**

|  |  |
| --- | --- |
| * Ipiece orientation, row, column = True if I piece is currently in play * Opiece orientation, row, column = True if O piece is currently in play * Jpiece orientation, row, column = True if J piece is currently in play * Lpiece orientation, row, column = True if L piece is currently in play * Spiece orientation, row, column = True if S piece is currently in play * Zpiece orientation, row, column = True if Z piece is currently in play * Tpiece orientation, row, column = True if T piece is currently in play |  |

# Constraints

* Let m and n be the size of the board where m is the width of the board and n is the height of the board.
* r must be within 0 and m - 1 inclusive
* c must be within 0 and n - 1 inclusive
* row and column must be within the grid
* Piece must fall after set amount of time

1. Round limit reached (Round)
   1. ~Round OR Round
   2. Certain number of rounds before the game will stop
2. Rotation locked (Locked)
   1. ~Locked OR Locked
   2. Since rotation and movement is locked right before it is placed to prevent sliding or T-Spins.
3. Piece Falling (Falling)
   1. ~Falling or Falling
   2. A piece will continue moving downwards until it is placed

* Multiple pieces can’t be in the same space
* Can only think about one piece placement at a time
* No universal gravity. This means when you clear line other lines don’t fall into slots below the cleared line

# Model Exploration

We currently haven’t implemented any code yet. We do, however, have a few ideas on how we could go about it and what should be kept in mind while we do so:

* Representing the Tetris grid as a 2D array, where a ‘0’ represents an empty space and an ‘x’ represents a filled space
  + Ex. Given a 4x6 grid, a potential 2D array could look like this:

0 0 0 0

0 0 x 0

x x x x

x x x x

x x 0 x

* Using a SATsolver to figure out the number of solutions given a certain condition. Potential conditions include:
  + Clear as many lines
  + Minimize height
  + Set the goal to certain score
  + Place a certain number of pieces
* Things to keep in mind:
  + Keeping piece at the top until you place (i.e. no falling)
  + Keeping track of the space around the piece
  + Delay rotation to prevent last moment rotation

# Jape Proof Ideas

* Given a 3x3 Tetris grid, a piece can only go where there’s room for it (?)
* If there’s a certain type of piece that can’t be placed on a full board, then it implies that another piece can’t be placed
* If you’re not currently placing the O piece, then it implies that you are placing one of the other types of pieces
  + A = O piece, B = I piece, C = J piece, D = L piece, S = S piece, F = Z piece, T = T piece
  + A, A**(**I J L S Z T) ⊢ (I J L S Z T)
* If the T piece is in orientation 1, then it is not in orientation 2, 3, or 4
  + T\_1, T\_1  **(**T\_2 T\_3 T\_4) ⊢ **(**T\_2 T\_3 T\_4)
* Show that the blocks will stay within the scope of the grid
* Have the jape proofs build onto each other
  + Specific board state

# First-Order Extension

An example of how our model could be extended to a predicate logic setting is the following:

Whether of not a piece can be placed in a certain spot:

Let A be the set of all coordinates on the board and let P be any subset of A of length k, where k is the number tiles in a certain type of piece. Depending on the type of piece, there is a certain model P must follow in order to fit the shape of the piece.

For example, a vertical 4 long piece must have a set P in the form {(i, j), (i + 1, j), (i + 2, j), (i + 3, j)}

Therefore, P is a valid placement if and only if:

* It is the correct piece type/shape
* any(((i + 1, j) in P) and ((i + 2, j) in P) and ((i + 3, j) in P) for i, j in P)
* Otherwise written as ∃i.∃j.(R(i, j)), where R(i, j) = P(i, j) ∧ P(i + 1, j) ∧ P(i + 2, j) ∧ P(i + 3, j)
* And there is an empty space that the piece can fit in
* all(!Bij for i, j in P)
* Otherwise written as ∀i.∀j.(¬B(i, j) ∨ ¬P(i, j))
* And considering (i, j) for each cell in the board, either the cell has to be in a different column, empty, or lower than this (i, j)
* all(all((i > r) or (j != c) or (!Br,c) for r, c in A) for i, j in P)
* Otherwise written as ∀i.∀j.∀x.∀y.(H(i, x) ∨ S(j, y) ∨ ¬B(x, y) ∨ ¬A(x, y)

∨ ¬P(i, j))

* This represents that you can't have a piece that’s placed below the blocks that are already there