Project Summary

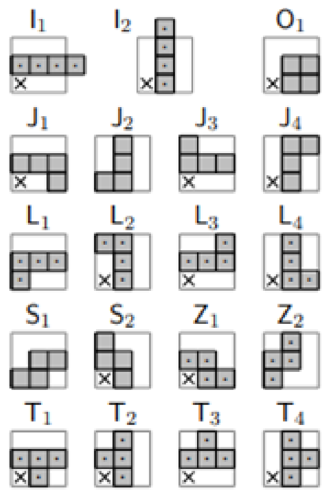
This project aims to figure out how many ways you can win a Tetris game given a certain win condition for the player. Our definition of a "win" is for the game to not end before a specified number of pieces have been placed. T-spins, which are last-minute rotations, and sliding will not be calculated as potential moves. We also not consider "row clearing". 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.

Note To Grader: The visualizer, which utilizes Pygame does not appear to work in the docker container due to the container environment lacking a display to create a window in. To see the visualizer please comment out the count_solutions and uncomment the visualizer and run the program in Windows/Mac.

Propositions

- **TetrisPiece** $_{\text{piece, rotation, x, y, time}}$**:** Piece of type ***piece*** (I, O, J, L, S, Z, or T), is placed with rotation number ***rotation***, with the top left corner placed at cell (***x, y***), (***x*** is the row number, ***y*** is the column number), at exactly round ***time***.
- **Cell** $_{\text{x, y, time}}$**:** The cell at **(*x, y*)** is occupied at round ***time*** (I.e., there was a piece placed at this cell, at round *time* or earlier)

(NOTE: Diagram is not fully accurate to how the pieces are represented in the model. The counting of the orientation types starts at 0 instead of 1. Also, the pieces are anchored to the top left corner instead of the bottom left shown and are modeled on a 4x4 grid.)



Constraints

Let **T** be TetrisPiece
Let **C** be Cell
Let **ROUNDS** be the total number of rounds
Let **ROWS** be the number of rows in the grid
Let **COLUMNS** be the number of columns in the grid
Let **p** be a piece type
Let **r** represent a rotation
Let **x** represent a row number
Let **y** represent a column number
Let **t** represent a round number
Let **(x0, y0), (x1, y1), (x2, y2) and (x3, y3)** be the cells that a piece occupies.

Note that cell rotation numbers, row indexes, column indexes and round numbers are 0 indexed.

Note that the top left corner is (0, 0), the bottom right corner is (ROWS – 1, COLUMNS – 1)

- There must be exactly one piece placed in each round

$$((T_{p, r, x, y, 0} \land \neg T_{p, r, x, y, 1} \land \neg T_{p, r, x, y, 2} \land \neg T_{p, r, x, y, 3} \land \ldots \land \neg T_{p, r, x, y, ROUNDS-1}) \lor$$
$$(\neg T_{p, r, x, y, 0} \land T_{p, r, x, y, 1} \land \neg T_{p, r, x, y, 2} \land \neg T_{p, r, x, y, 3} \land \ldots \land \neg T_{p, r, x, y, ROUNDS-1}) \lor$$
$$\ldots \lor$$
$$(\neg T_{p, r, x, y, 0} \land \neg T_{p, r, x, y, 1} \land \neg T_{p, r, x, y, 2} \land \neg T_{p, r, x, y, 3} \land \ldots \land T_{p, r, x, y, ROUNDS-1}))$$

- A piece that is placed must fall onto the bottom of the grid or on to another piece. If x0 = 0 or x1 = 0 or x2 = 0 or x3 = 0, then the piece is on the bottom. If the piece is not on the bottom, then we must ensure that there was a cell occupied at t-1 below one of the cells that the new piece will occupy. If t = 0 then we must add the constraint $\neg T_{p, r, x, y, 0}$, as there is no previous time. Otherwise, we add the full constraint:

$T_{p, r, x, y, t} \longrightarrow (C_{x0+1, y0, t-1} \vee C_{x1+1, y0, t-1} \vee C_{x2+1, y0, t-1} \vee C_{x3+1, y0, t-1})$

- If a piece is placed, the cells that it would take up depending on its piece type, rotation and location, must have a block in them at the time this piece was placed, and every later time.

$$\begin{aligned} T_{p, r, x, y, t} \longrightarrow ((&C_{x0, y0, t} \wedge C_{x0, y0, t+1} \wedge C_{x0, y0, t+2} \wedge C_{x0, y0, ROUNDS-1}) \wedge \\ &\ldots \wedge \\ (&C_{x3, y3, t} \wedge C_{x3, y3, t+1} \wedge C_{x3, y3, t+2} \wedge C_{x3, y3, ROUNDS-1})) \end{aligned}$$

- A piece may not travel through a previously placed piece (or overlap with). This can be enforced by ensuring that if a piece is placed at a certain time, then for all the cells that this piece occupies, there are not any cells occupied previously in the same column at this row or higher. Constraint is not enforced if t = 0.

$$\begin{aligned} T_{p, r, x\, y, t} \longrightarrow ((&\neg C_{x0, y0, t-1} \wedge \neg C_{x0-1, y0, t-1} \wedge \neg C_{x0-2, y0, t-1} \wedge \ldots \wedge \neg C_{0, y0, t-1}) \wedge \\ &\ldots \wedge \\ (&\neg C_{x3, y0, t-1} \wedge \neg C_{x3-1, y0, t-1} \wedge \neg C_{x3-2, y0, t-1} \wedge \ldots \wedge \neg C_{0, y0, t-1})) \end{aligned}$$

- There must not be any cells with a block in them that are outside of the grid.

$\neg C_{ROWS, COLUMNS, t} \wedge \neg C_{ROWS + 1, COLUMNS, t} \wedge \neg C_{ROWS, COLUMNS + 1, t} \wedge \ldots$

- A piece must be placed for a cell to be blocked. Given that each piece has 4 blocks, and one piece is placed at each ***time***, this constraint can be enforced by ensuring that are not more 4 * (***time*** + 1) cells blocked at each ***time***. Or equivalently, there are at least ROWS * COLUMNS – 4 * (***time*** + 1) cells that are not blocked. Let b = ROWS * COLUMNS – 4 * (***time*** + 1). Let i be a single index number to replace (x, y), i = COLUMNS * x + y

$$\begin{aligned} ((&\neg C_{0, t} \wedge \neg C_{1, t} \wedge \ldots \wedge \neg C_{b-1, t}) \vee (\neg C_{1, t} \wedge \neg C_{2, t} \wedge \ldots \wedge \neg C_{b, t}) \vee \ldots \\ &\vee (\neg C_{ROWS*COLUMNS-b, t} \wedge \neg C_{ROWS*COLUMNS-b+1, t} \wedge \ldots \wedge \neg C_{ROWS*COLUMNS-1, t})) \end{aligned}$$

Model Exploration

Initial Model

The original idea is just modelling the game of Tetris, however with feedback from Prof. Muise we were informed that we need to include the element of time but remove “sliding” and “T-spin” like moves. After the initial modeling we should attempt to explore different ways the SAT solver can work (random moves, minimize height, etc..).

Propositions of pieces were originally numbers 0-$19_{r,c}$ with 0-3 as Z piece and so on, now we have decided to separate Tetris pieces into different propositions with the orientation, position and time.

Originally location had row and column, however, to include time we will replace row location with time and have the block constantly move down with time.

Initial Steps:

In the beginning we had a very rough idea of how to model Tetris, the majority of the ideas in this model were scrapped after feedback from our peers and very little or none was brought into the final implementation.

- Tetris Blocks not separated
    - 0-19 with subsets denoting Tetris piece types
    - Many different propositions
        - Row clearing
        - Blocked
        - Fail
    - Constraints
        - Rotation Locking
        - Falling piece
        - Pieces can’t be placed in same place
    - No universal gravity, placed blocks float

Overall, our initial model was from perspective that attempted to model Tetris in a way that a programmer trying to program the game Tetris itself. This can be seen in the way we were attempted to thinking about rotations in the present rather than viewing the rotations as a set of possible placements.

Intermediate Models

After attempting to create a SAT solver we realize that we were generating propositions and constraints in a way more appropriate for coding a game. We then adjusted propositions and constraints to suit the Bauhaus library model better. To do this we knew we needed to reduce the propositions. To do this we only focused on the core of Tetris which related to the Tetris pieces themself. Everything else was dropped from the propositions. During the intermediate stages of our model exploration narrowed down proposition(s) changed a few times.

1.
    a. TetrisPiece(roundNumber, type, orientation, location, time)

This idea we only had one massive proposition that was meant to consider everything in one fell swoop. This may have been possible to properly implement, however this model was a first attempt at implementation and was made with familiarizing ourselves with the encoding tools we were provided.

2.
    a. PieceConfig(type, orientation)
    b. PiecePlacement(piececonfig, location, time)

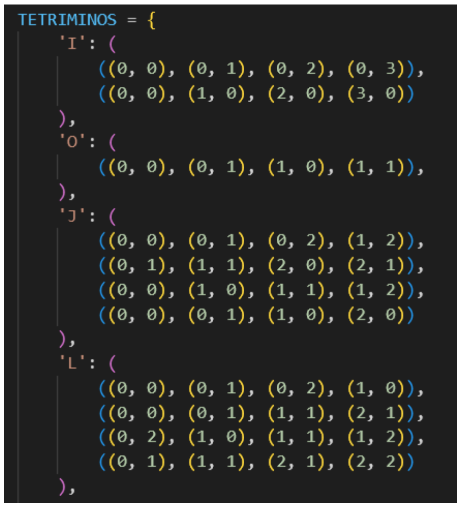
With more familiarity with the encoding process, we decided to reference Prof. Muise's Kanoodle project as the two games are similar. We wanted to see what parts of Kanoodle could be shared with Tetris, through this we were able to develop a working model of Tetris.

3.
    a. TetrisPiece(type,rotation,x,y,time)
    b. Cell(x,y,time)

After learning a bit about how Prof. Muise modeled kanoodle we figured out some elements that we could share. We then decided on the two propositions above, adding the element of time through the existence of rounds we decided to have the propositions representing the Tetris piece in play on a given round and the state of each cell on a round. With the cell propositions we figured out that we could represent all previous rounds, in a sense the initial model's blocked proposition evolved into our cell proposition.

Final Model

Constants:



```
TETRIMINOS = {
    'I': (
        ((0, 0), (0, 1), (0, 2), (0, 3)),
        ((0, 0), (1, 0), (2, 0), (3, 0))
    ),
    'O': (
        ((0, 0), (0, 1), (1, 0), (1, 1)),
    ),
    'J': (
        ((0, 0), (0, 1), (0, 2), (1, 2)),
        ((0, 1), (1, 1), (2, 0), (2, 1)),
        ((0, 0), (1, 0), (1, 1), (1, 2)),
        ((0, 0), (0, 1), (1, 0), (2, 0))
    ),
    'L': (
        ((0, 0), (0, 1), (0, 2), (1, 0)),
        ((0, 0), (0, 1), (1, 1), (2, 1)),
        ((0, 2), (1, 0), (1, 1), (1, 2)),
        ((0, 1), (1, 1), (2, 1), (2, 2))
    ),
```

Whilst creating our final model we continued to consult the feedback provided to us by our peers. One of which recommended to model our piece proposition by the blocks it occupies. We decided to implement this in our code through a TETRIMINOS dictionary.

Propositions:

- TetrisPiece(type,rotation,x,y,time)

- Cell(x,y,time)

Constraints:

- There must be only one Tetris piece placed in each round
- For all pieces, when placed they will then occupy those cells
    - The piece must "fall" to the bottom of the grid
- No piece can be placed out of bounds
- For a cell to be occupied, a piece must have been placed in that location

After learning how to use the Bauhaus library and experimenting with different propositions, we eventually settled on a model we thought could work. We believe these are the two basic propositions needed to model Tetris. With this we could now create constraints that would form the rules of Tetris, before this we simply had a model which could place Tetris pieces on a grid without rules, meaning that they could overlap, go out of bounds, phase through other pieces and so on. The constraints are meant to ground this model to the rules of Tetris.

Model Counting

There were different constants that we tried to use:

- 2x2, 3x2, 3x3, 4x3, 4x4, 5x5
- 1 round & 2rounds

We found that with Bauhaus and the docker desktop we were extremely limited in the solutions we could count. In a Linux environment the maximum board size we could have without the program crashing was 4x3 with 1 round or 4x2 with 2 rounds.

Functions

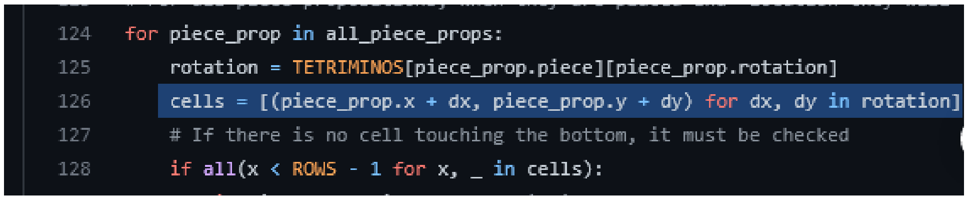
- E.satisfiable(): This function shows whether a solution exists to a given model, this allowed us to get a quick understanding of whether there were any issues with our model as we were building the constraints. For example, when Model Counting if we had an L piece in a 2x2 if satisfiable returned true we would know there was something fundamentally wrong with the constraints.

- Count_solutions(E): This function allowed us to further validate our models' constraints during the implementation process. As you will see below in the Debugging section, this function helped us gain more detail into what was errors that were present in the constraints but were not as obvious as true and false.

- Visualizer: This is a function to create visual representation of one of the solutions found in a model, this is quick tool that gives us a quick and intuitive way to check how pieces are being "placed" in the model.

Debugging

At some point we noticed that there we an absurd number of solutions despite having an extremely simple model. For example, an O piece in a 3x2 grid would result in 52 solutions. The issue was a bug in the code which ended up being:



```
124 for piece_prop in all_piece_props:
125     rotation = TETRIMINOS[piece_prop.piece][piece_prop.rotation]
126     cells = [(piece_prop.x + dx, piece_prop.y + dy) for dx, dy in rotation]
127     # If there is no cell touching the bottom, it must be checked
128     if all(x < ROWS - 1 for x, _ in cells):
```

The bug was that a generator was used on line 126 instead of a list, which when the program attempted to iterate over the generator it picked up where it left off every time instead of starting from the beginning as intended. To fix this we changed the generator to a list so we could iterate over all the values more than once. This resolved the bug and the same test example resulted in the correct 2 solutions.

First-Order Extension

Predicates:

The first two will mirror closely to the propositions defined above.

- Let **T** be TetrisPiece(p, r, x, y, t)
- Let **C** be Cell(x, y, t)
- **Z(r)** takes a round/row number and returns true if r = 0, otherwise false
- **L(x0, x1)** takes two row numbers and returns true if x0 < x1, otherwise false

Variables

- Let **p** be a piece type
- Let **r** be rotation
- Let **x** is a row number
- Let **y** is a column number
- Let t be a round number
- Let **(xi, yi)** be one cell that a piece occupies

Functions:

- I(r) takes a round/row number and returns r + 1

- D(r) takes a round/row number and return r - 1

Constraints (from propositional constraints):

- A piece may not travel through a previously placed piece (or overlap with). This can be enforced by ensuring that if a piece is placed at a certain time, then for all the cells that this piece occupies, there are not any cells occupied previously in the same column at this row or higher. Constraint is not enforced if t = 0.

Predicate logic constraint:

$$\forall p.\forall r.\forall x.\forall y.\forall t.\left(T(p,r,x,y,t) \rightarrow Z(t) \vee \forall i.\forall x.\left(L(xi,x) \vee \backsim C\left(x,yi,D(t)\right)\right)\right)$$

- A piece that is placed must fall onto the bottom of the grid or on to another piece. If x0 = 0 or x1 = 0 or x2 = 0 or x3 = 0, then the piece is on the bottom. If the piece is not on the bottom, then we must ensure that there was a cell occupied at t-1 below one of the cells that the new piece will occupy. If t = 0 then we must add the constraint ¬ $T_{p, r, x, y, 0}$, as there is no previous time. Otherwise, we add the full constraint:

Predicate logic constraint:

$$\forall p.\forall r.\forall x.\forall y.\forall t\left(T(p,r,x,y,t) \rightarrow \exists i.Z(xi) \vee \left(\backsim Z(t) \wedge \exists i.C\left(I(xi),yi,D(t)\right)\right)\right)$$

Jape Proofs

In these proofs, a 2x2 grid and a pair of pieces are used in order to simplify the model. Also, we assume that the piece placement is occurring during the same round. So, time is constant.

Propositions Used

- **A xy** – Piece 1 is at location x,y on grid
- **B xy** – Piece 2 is at location x,y on grid
- **C xyz** – Cell grid is filled in at location x,y with piece z

Base premises

When piece is placed, it implies that the cell in filled

- $A11 \rightarrow (C111 \wedge C211)$
- $A12 \rightarrow (C121 \wedge C221)$
- $B11 \rightarrow (C112 \wedge C212)$
- $B12 \rightarrow (C122 \wedge C222)$

A cell can’t be filled by two pieces

- $\neg(C111 \wedge C112)$
- $\neg(C121 \wedge C122)$
- $\neg(C211 \wedge C212)$
- $\neg(C221 \wedge C222)$

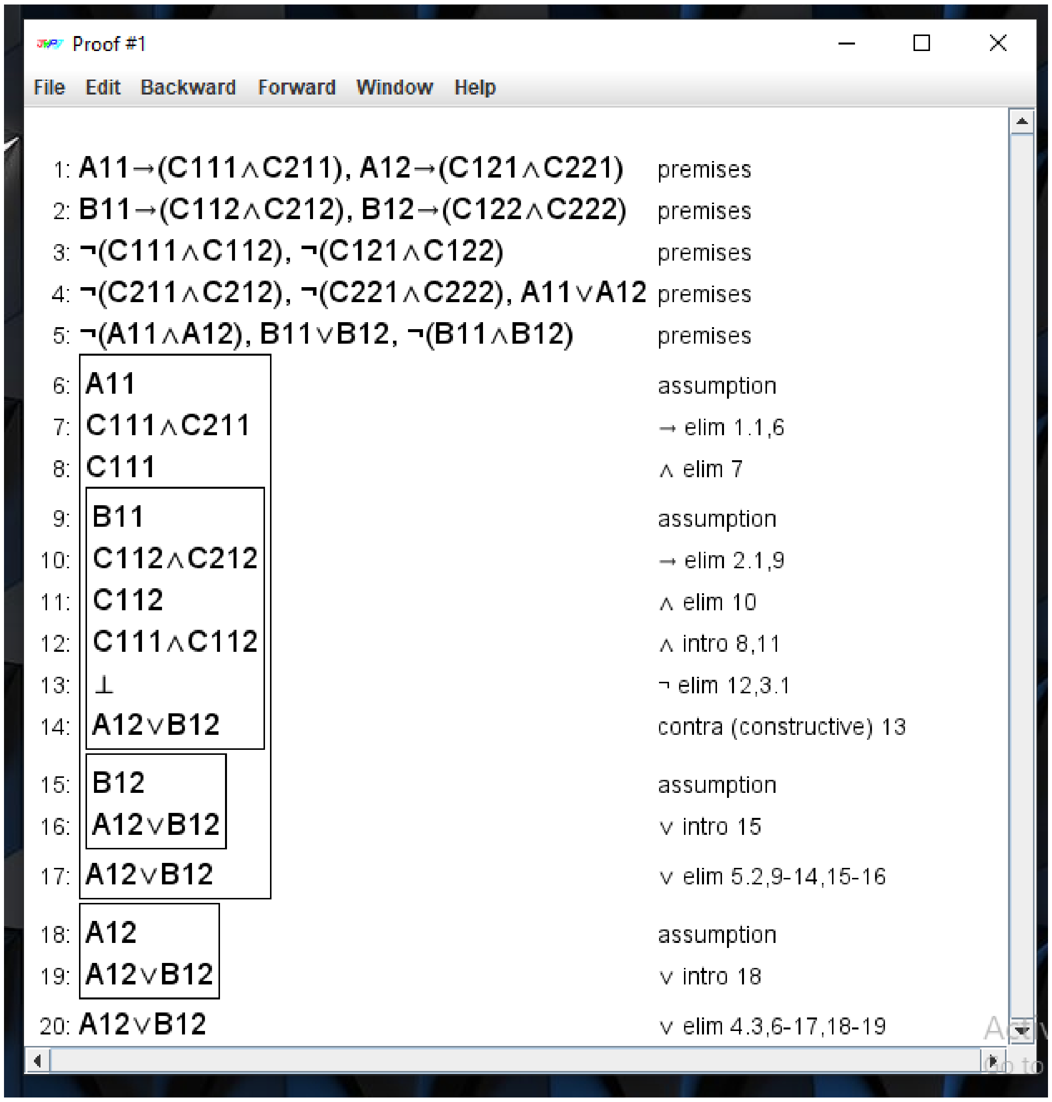
Each piece is placed somewhere

- $A11 \vee A12$
- $\neg(A11 \wedge A12)$
- $B11 \vee B12$
- $\neg(B11 \wedge B12)$

Proof #1

When placing Piece A and Piece B, either piece must fill cell (1,2)

- (Base encoding) ⊢ A12∨B12

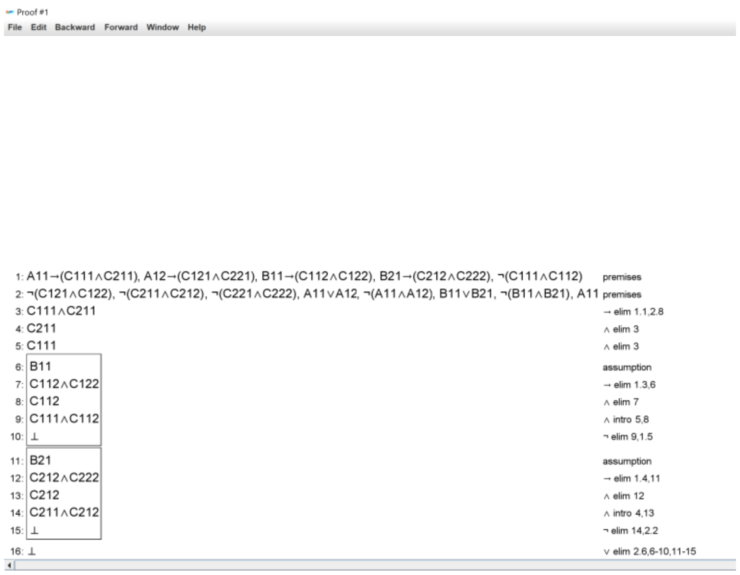


```
 1: A11→(C111∧C211), A12→(C121∧C221)  premises
 2: B11→(C112∧C212), B12→(C122∧C222)  premises
 3: ¬(C111∧C112), ¬(C121∧C122)  premises
 4: ¬(C211∧C212), ¬(C221∧C222), A11∨A12  premises
 5: ¬(A11∧A12), B11∨B12, ¬(B11∧B12)  premises
 6:   A11  assumption
 7:   C111∧C211  → elim 1.1,6
 8:   C111  ∧ elim 7
 9:     B11  assumption
10:     C112∧C212  → elim 2.1,9
11:     C112  ∧ elim 10
12:     C111∧C112  ∧ intro 8,11
13:     ⊥  ¬ elim 12,3.1
14:     A12∨B12  contra (constructive) 13
15:     B12  assumption
16:     A12∨B12  ∨ intro 15
17:   A12∨B12  ∨ elim 5.2,9-14,15-16
18:   A12  assumption
19:   A12∨B12  ∨ intro 18
20: A12∨B12  ∨ elim 4.3,6-17,18-19
```

Proof #2

If pieces can’t fit on grid without overlapping, then piece can’t be placed/ things can’t be placed out of bounds. Piece A is already placed at (1,1) on the grid

- (Base encoding, but Piece B is horizontal), A11 ⊢ ⊥
  - B11→(C112∧C122)
  - B21→(C212∧C222)



```
 1: A11→(C111∧C211), A12→(C121∧C221), B11→(C112∧C122), B21→(C212∧C222), ¬(C111∧C112)   premises
 2: ¬(C121∧C122), ¬(C211∧C212), ¬(C221∧C222), A11∨A12, ¬(A11∧A12), B11∨B21, ¬(B11∧B21), A11   premises
 3: C111∧C211                 → elim 1.1,2.8
 4: C211                      ∧ elim 3
 5: C111                      ∧ elim 3
 6: | B11                     assumption
 7: | C112∧C122               → elim 1.3,6
 8: | C112                    ∧ elim 7
 9: | C111∧C112               ∧ intro 5,8
10: | ⊥                       ¬ elim 9,1.5
11: | B21                     assumption
12: | C212∧C222               → elim 1.4,11
13: | C212                    ∧ elim 12
14: | C211∧C212               ∧ intro 4,13
15: | ⊥                       ¬ elim 14,2.2
16: ⊥                         ∨ elim 2.6,6-10,11-15
```

Proof #3

For a piece to be placed, there must be a clear vertical path. Therefore, in a 2x2 grid, with 2 horizontal pieces, the first piece must be placed at the bottom and the second at the top/

- (Base encoding but Piece A and B are not horizontal), ¬(C121∧C221) ∨ ¬(C122∧ C222) ⊢ (C111 ∧ C211) ∨ (C112 ∧ C212)
- A11 → (C111 ∧ C121)
- A21 → (C211 ∧ C221)
- B11 → (C112 ∧ C122)
- B21 → (C212 ∧ C222)
- C111 → ¬C212: If piece 1 is placed in the top left corner, then piece 2 cannot be placed below it, in the bottom left corner
- C121 → ¬C222: If piece 1 is placed in the top right corner, then piece 2 cannot be placed below it, in the bottom right corner

```
18: A21∧B11                                        ∨ elim 2.5,7-12,13-17
31: A21∧B11                                        ∨ elim 2.3,3-18,19-30
```