GROUP 14: Unwordle

Madison MacNeil (20mkm17)

Simon Nair (21scn1)

Ryan Van Drunen (21rdbv)

Course Modelling Project

CISC/CMPE 204

Logic for Computing Science

December 8, 2023

Abstract

The goal of this project is to model a rendition of the game Wordle. Wordle is a word guessing game where a correct 5 letter word is chosen, and feedback is given on each player guess. Green tiles represent a letter in the right spot, yellow tiles represent a letter in the wrong spot, and white or grey tiles represent a letter not in the final word. The player's guesses must also be a valid word.

Our model is given a board configuration - the colours of each position in the board in a 2D array - as well as a final word, and is tasked to find all possible solutions, while displaying one possibility.

Propositions

This model only has a few propositions.

1. $\textbf{Tile}_{x,y,c}$: represents a tile at some index x, y with a color c
2. $\textbf{Letter}_{x,y,c,l}$: represents a letter l at some index x, y with a colour c
3. $\textbf{Row}_{n,l1,l2,l3,l4,l5}$: represents a row in the y position n, containing the letters $l1, l2, l3, l4, l5$
4. $\textbf{Board}_{r1,r2,r3,r4}$: represents a set of 4 rows, $r1, r2, r3, r4$, that compose a potential board

Constraints

1. **Green Letter Constraint:** A letter can be green at some index x, y only if that letter is in the solution at the same x index.
   - **Letter(r, col, letter, "Green")** >> **Letter(3, col, letter, "Green")** for all r, col.
2. **Yellow Letter Exclusion Constraint:** A letter cannot be yellow at some index x, y if that letter is in the solution at the same x index.
   - **Letter(3, col, letter, "Green")** >> ¬**Letter(r, col, letter, "Yellow)** for all r, col.
3. **Yellow Letter Inclusion Constraint:** A letter can only be yellow at some index x, y if it is a letter in the solution.
   - **Letter(r, col, letter, "Yellow")** >> **(Letter(3, 0, letter, "Green")** ∨
     Letter(3, 1, letter, "Green") ∨
     Letter(3, 2 ,letter, "Green") ∨
     Letter(3, 3, letter, "Green") ∨
     Letter(3, 4, letter, "Green"))
     for all r, col.
4. **White Letter Constraint:** A White letter at any index x, y cannot be a letter in the solution.

- **Letter(r, col, letter, "White") >> ¬Letter(3, col2, letter, "Green)** for all r, col, col2.

5. **Letter Colour Constraint:** A letter at some index x, y can only be the colour of tile at that index.

   - **Letter(r, col, letter, colour) >> Tile(r, col, colour)** for all r, col, letter, colour.

6. **Row Constraint:** A row must be made up of 5 True Letter Propositions

   - **Letter(r, 0, l1, c1) ∧**
     Letter(r, 1, l2, c2) ∧
     Letter(r, 2, l3, c3) ∧
     Letter(r, 3, l4, c4) ∧
     Letter(r, 4, l5, c4) >> Row(num, l1, l2, l3, l4, l5)
     for all r, c(1-5), l(1-5) and num.

7. **Board Constraint:** A board must be composed of 4 True Row Propositions

   - **Row(1, l1, l2, l3, l4, l5) ∧**
     Row(2, l1, l2, l3, l4, l5) ∧
     Row(3, l1, l2, l3, l4, l5) ∧
     Row(4, l1, l2, l3, l4, l5) >> Board(r1,r2,r3)
     for all r, c(1-5), each unique l(1-5) in a row.

Model Exploration

Our model as it stands today was born largely through the process of trial and error, as we came to understand the Bauhaus Library, alongside the many nuances of representing a large environment with logic.

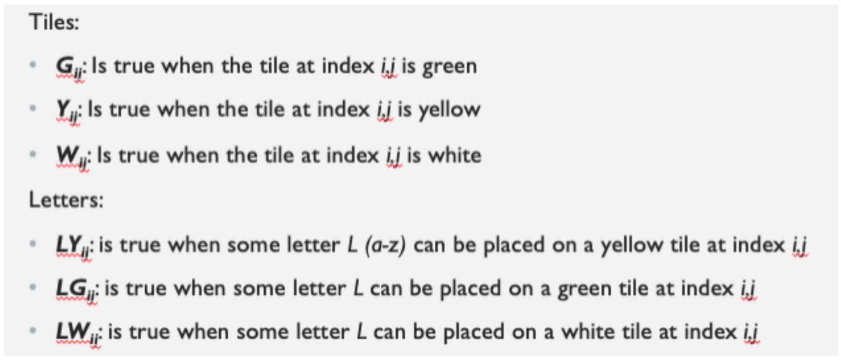
Creating Representative Propositions

The propositions we created in our initial project proposal failed to adequately represent the relationship between some object (letter or tile) and the colour of that object. The colour of the object had been baked into the variable name rather than as an attribute of that variable. These initial propositions can be seen below in Figure 1.

The nature of these propositions meant that letter and tile objects could not be created in a general fashion by iterating through all x, y positions, all letters and all colours. This presented problems in the formulation of constraints.

We resolved this issue by establishing a $\text{Tile}_{(x,y,c)}$ proposition that represented some position on the empty board at x, y and its colour, c and an $\text{Assigned}_{(t,l,c)}$ proposition representing some colour c, letter l existing at some tile t. The updated propositions are available in Figure 2.

As we began developing our constraints we were able to refine our propositions into one proposition $\text{Tile}_{(x,y,l,c)}$ that represented some letter l, of some colour c at some position x, y. This proposition is available in Figure 3:



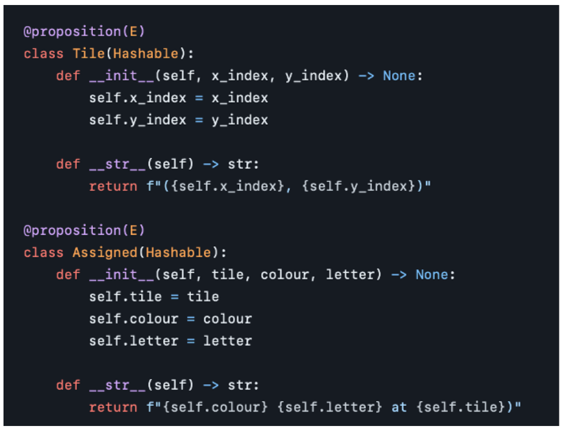
Tiles:

- G_{ij}: Is true when the tile at index *i,j* is green
- Y_{ij}: Is true when the tile at index *i,j* is yellow
- W_{ij}: Is true when the tile at index *i,j* is white

Letters:

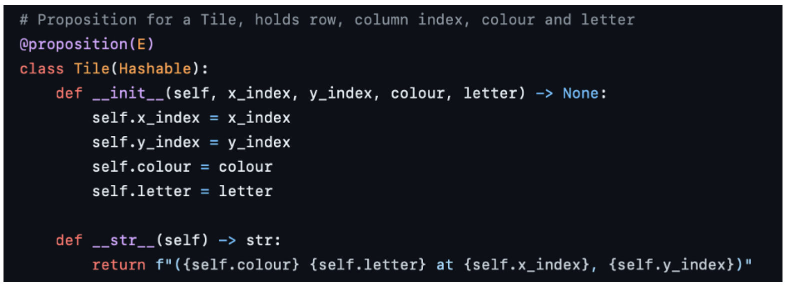
- LY_{ij}: is true when some letter *L (a-z)* can be placed on a yellow tile at index *i,j*
- LG_{ij}: is true when some letter *L* can be placed on a green tile at index *i,j*
- LW_{ij}: is true when some letter *L* can be placed on a white tile at index *i,j*

Figure 1: Project Proposal Propositions



```python
@proposition(E)
class Tile(Hashable):
    def __init__(self, x_index, y_index) -> None:
        self.x_index = x_index
        self.y_index = y_index

    def __str__(self) -> str:
        return f"({self.x_index}, {self.y_index})"

@proposition(E)
class Assigned(Hashable):
    def __init__(self, tile, colour, letter) -> None:
        self.tile = tile
        self.colour = colour
        self.letter = letter

    def __str__(self) -> str:
        return f"{self.colour} {self.letter} at {self.tile})"
```

Figure 2: Second Formulation of Propositions

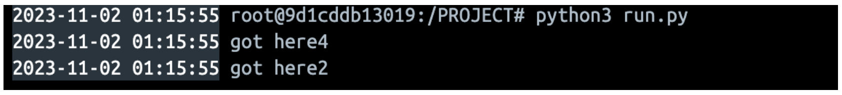


```python
# Proposition for a Tile, holds row, column index, colour and letter
@proposition(E)
class Tile(Hashable):
    def __init__(self, x_index, y_index, colour, letter) -> None:
        self.x_index = x_index
        self.y_index = y_index
        self.colour = colour
        self.letter = letter

    def __str__(self) -> str:
        return f"({self.colour} {self.letter} at {self.x_index}, {self.y_index})"
```

Figure 3: Third Proposition Formulation

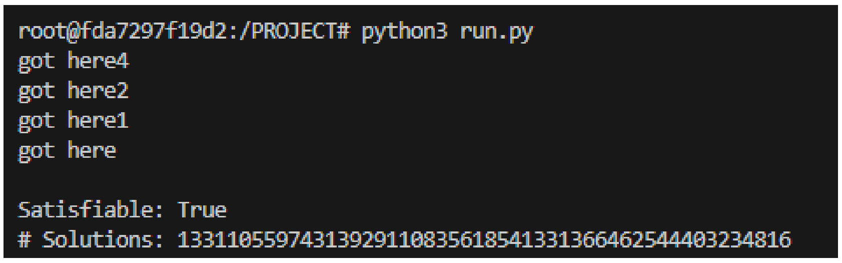
Error Location

As we were developing the model we encountered a number of errors, most notably the program being killed by a run-length error, or internal Bauhaus errors we did not yet well understand. As these errors persisted, and program was yet to reach the point of creating any solutions, we introduced output flags in our code that would allow us to tell which parts of the program had been reached and had run before the error occurred. Of the four flags we introduced, one was placed after the creation of our propositions, one was placed within our build theory, one after the build theory was called and the last was placed after compilation. An example of the output flags when the program did not run successfully can be seen in Figure 4 and, contrarily, an example of the output flags when entirety of program ran successfully can be seen in Figure 5.



```
2023-11-02 01:15:55 root@9d1cddb13019:/PROJECT# python3 run.py
2023-11-02 01:15:55 got here4
2023-11-02 01:15:55 got here2
```

Figure 4: Output Flags for Unsuccessful Program Execution

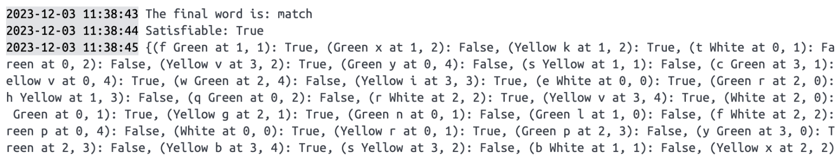


```
root@fda7297f19d2:/PROJECT# python3 run.py
got here4
got here2
got here1
got here

Satisfiable: True
# Solutions: 133110559743139291108356185413313664625444403234816
```

Figure 5: Output Flags for Successful Program Execution

Producing Meaningful Output

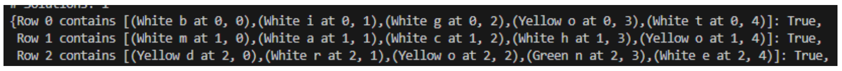
Once we achieved a model that was returning solutions, our immediate next step was to format the solution output for the sake of debugging. A small sample of the initial output of the solver, that was extensive and nearly impossible to parse through for errors, can be seen below in Figure 6:



```
2023-12-03 11:38:43 The final word is: match
2023-12-03 11:38:44 Satisfiable: True
2023-12-03 11:38:45 {(f Green at 1, 1): True, (Green x at 1, 2): False, (Yellow k at 1, 2): True, (t White at 0, 1): Fa
reen at 0, 2): False, (Yellow v at 3, 2): True, (Green y at 0, 4): False, (s Yellow at 1, 1): False, (c Green at 3, 1):
ellow v at 0, 4): True, (w Green at 2, 4): False, (Yellow i at 3, 3): True, (e White at 0, 0): True, (Green r at 2, 0):
h Yellow at 1, 3): False, (q Green at 0, 2): False, (r White at 2, 2): True, (Yellow v at 3, 4): True, (White at 2, 0):
 Green at 0, 1): True, (Yellow g at 2, 1): True, (Green n at 0, 1): False, (Green l at 1, 0): False, (f White at 2, 2):
reen p at 0, 4): False, (White at 0, 0): True, (Yellow r at 0, 1): True, (Green p at 2, 3): False, (y Green at 3, 0): T
reen at 2, 3): False, (Yellow b at 3, 4): True, (s Yellow at 3, 2): False, (b White at 1, 1): False, (Yellow x at 2, 2)
```

Figure 6: Sample of Solver Output - Unadjusted for Readability

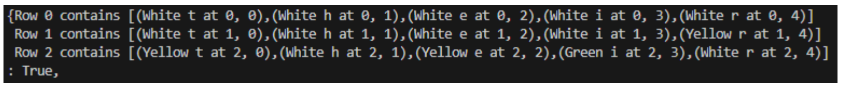
We chose to output solutions by rows, as we felt we would be able to observe the most information about the results of the solver in that form. We were able to observe which words were being used from our 5 letter word list, which letters were being assigned to which colours and at what indices. A sample of two row solution outputs are available in Figure 7:



```
{Row 0 contains [(White b at 0, 0),(White i at 0, 1),(White g at 0, 2),(Yellow o at 0, 3),(White t at 0, 4)]: True,
 Row 1 contains [(White m at 1, 0),(White a at 1, 1),(White c at 1, 2),(White h at 1, 3),(Yellow o at 1, 4)]: True,
 Row 2 contains [(Yellow d at 2, 0),(White r at 2, 1),(Yellow o at 2, 2),(Green n at 2, 3),(White e at 2, 4)]: True,
```

Figure 7: Two Row Solution Outputs

After exploring the output of the row solutions, we pushed further and formatted the solution output by single boards instead, which represented a combination of 3 rows, meant to satisfy the model. A sample of one such output can be seen in Figure 8:



```
{Row 0 contains [(White t at 0, 0),(White h at 0, 1),(White e at 0, 2),(White i at 0, 3),(White r at 0, 4)]
 Row 1 contains [(White t at 1, 0),(White h at 1, 1),(White e at 1, 2),(White i at 1, 3),(Yellow r at 1, 4)]
 Row 2 contains [(Yellow t at 2, 0),(White h at 2, 1),(Yellow e at 2, 2),(Green i at 2, 3),(White r at 2, 4)]
: True,
```

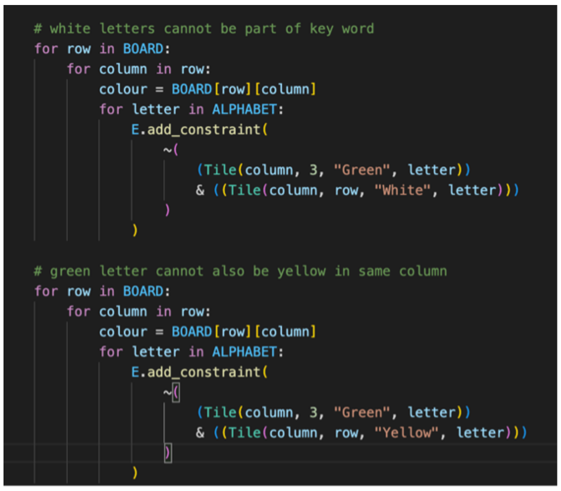
Figure 8: Solution Output Formatted As Boards

Development of a "Base-Case" of Solution Letters

After adjusting our solution output, we were able to observe a number of logical errors that were occurring as result of improperly implemented constraints. A non-comprehensive list of some of the errors we observed:

- Letters appearing as green, white and yellow in different positions on the board (Seen in Figure 8)
- The solution letters appearing in white board positions
- Rows containing 5 letters that should not have all been treated as valid in their respective positions
- Boards containing words that appear in more than one row (Seen in Figure 8 - the word "their" used in rows 0 - 2)

We came to understand that root of most, if not all, of the issues we were observing was a fundamental flaw in the manner in which our constraints were structured. Take for example the two constraints depicted in Figure 9, the first of which represents that some letter cannot be white if it is part of the solution word, and the second that if a letter cannot be yellow in the same column it appears as part of the solution word.



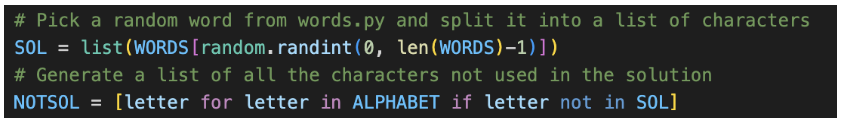
```python
# white letters cannot be part of key word
for row in BOARD:
    for column in row:
        colour = BOARD[row][column]
        for letter in ALPHABET:
            E.add_constraint(
                ~(
                    (Tile(column, 3, "Green", letter))
                    & ((Tile(column, row, "White", letter)))
                )
            )

# green letter cannot also be yellow in same column
for row in BOARD:
    for column in row:
        colour = BOARD[row][column]
        for letter in ALPHABET:
            E.add_constraint(
                ~(
                    (Tile(column, 3, "Green", letter))
                    & ((Tile(column, row, "Yellow", letter)))
                )
            )
```

Figure 9: Two Sample Constraints

The issue however, with these constraints is that no where in our model did we establish that a proposition representing a letter as green at some index in row 3, MUST be true if is in fact the letter that appears at that index in the solution word. As such, the model produces a series of solutions in which it arbitrarily decides the truth of a letter being green in the final row at any index, and the consequent truth of that letter being white or yellow anywhere else in the board, and continues a produce a model in line with the truth values thereof. Ultimately, the solver was producing a model for every iteration of those potential truth values, not what we were striving towards.

Splitting the Alphabet

By the realizations above, we proceeded to make significant changes to our implementation. On of the most notable changes was rather than approaching all the letters of the alphabet simultaneously, we chose to approach each letter as a member of one of two distinct groups. A letter was now either part of the solution word or not part of the solution word. Our implementation of splitting of letters as either in the solution or not, can be seen below in Figure 10:



```
# Pick a random word from words.py and split it into a list of characters
SOL = list(WORDS[random.randint(0, len(WORDS)-1)])
# Generate a list of all the characters not used in the solution
NOTSOL = [letter for letter in ALPHABET if letter not in SOL]
```

Figure 10: Dividing Alphabet Letters Into "In Solution" Array or "Not In Solution" Array

Having isolated all letters as members of one of two classes, and including them in the appropriate array, developing the constraints became much easier. We iterated through all the letters in the solution first, implementing the following constraints:

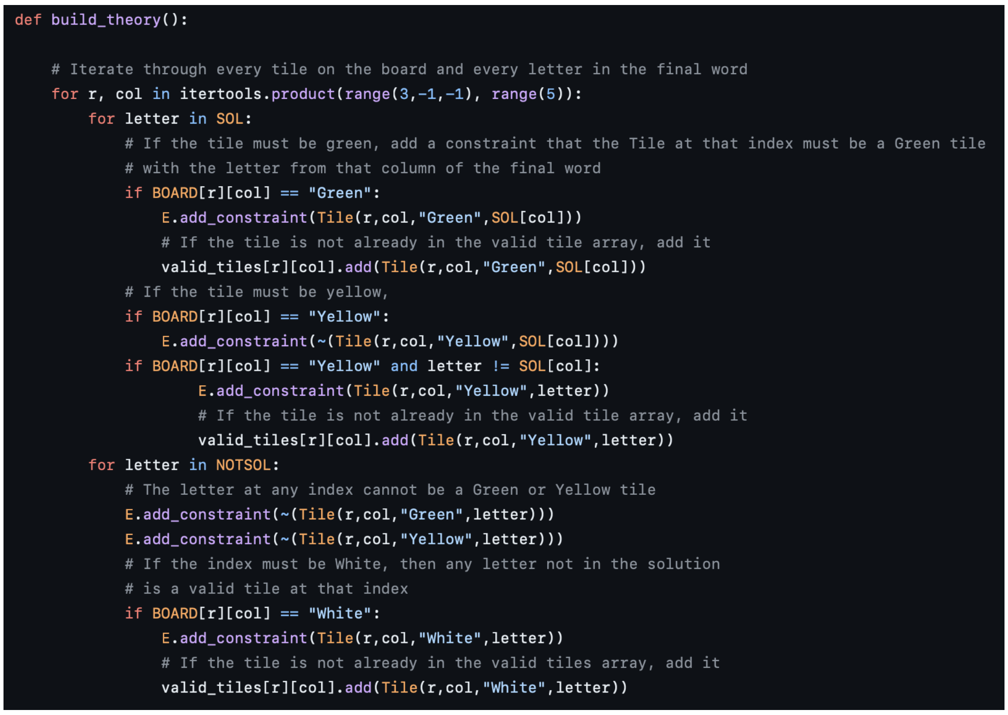
- Where a tile is green, the only true letter at that tile is the letter in the solution at the corresponding x index being green.
- Where a tile is yellow, it is always false that the letter in the solution at that x index is yellow.
- Where a tile is yellow, all letters in the solution word not at the x index of the tile in question are true when yellow.

Thereafter, we iterated through the letters not in the solution adding constraints that those letters can never be assigned to the colours yellow or green at any index in a proposition, and can only ever be assigned to the colour white when the board at that position is also white.

Initially, we were very pleased with the implementation found in Figure 11, as it was the first we had created that was produced letter assignments that were entirely correct. A sample of the letter assignments for this implementation are available in Figure 12.

The letter assignments in Figure 12 are correct for the final word and board layout in question, however a key point to realize is that the solution of all possible letter assignments output as one solution (See "Solutions: 1"), rather than multiple solutions that each represent some permutation of true letter assignments.

We kept this implementation for sometime, and continued on to develop constraints for the Row and Board propositions. We were able to create a solver that produced valid solutions for



```python
def build_theory():

    # Iterate through every tile on the board and every letter in the final word
    for r, col in itertools.product(range(3,-1,-1), range(5)):
        for letter in SOL:
            # If the tile must be green, add a constraint that the Tile at that index must be a Green tile
            # with the letter from that column of the final word
            if BOARD[r][col] == "Green":
                E.add_constraint(Tile(r,col,"Green",SOL[col]))
                # If the tile is not already in the valid tile array, add it
                valid_tiles[r][col].add(Tile(r,col,"Green",SOL[col]))
            # If the tile must be yellow,
            if BOARD[r][col] == "Yellow":
                E.add_constraint(~(Tile(r,col,"Yellow",SOL[col])))
            if BOARD[r][col] == "Yellow" and letter != SOL[col]:
                    E.add_constraint(Tile(r,col,"Yellow",letter))
                    # If the tile is not already in the valid tile array, add it
                    valid_tiles[r][col].add(Tile(r,col,"Yellow",letter))
        for letter in NOTSOL:
            # The letter at any index cannot be a Green or Yellow tile
            E.add_constraint(~(Tile(r,col,"Green",letter)))
            E.add_constraint(~(Tile(r,col,"Yellow",letter)))
            # If the index must be White, then any letter not in the solution
            # is a valid tile at that index
            if BOARD[r][col] == "White":
                E.add_constraint(Tile(r,col,"White",letter))
                # If the tile is not already in the valid tiles array, add it
                valid_tiles[r][col].add(Tile(r,col,"White",letter))
```

Figure 11: First Successful Implementation

our model, however we felt that we had sold ourselves short in the manner in which we had represented the letter assignments using logic.

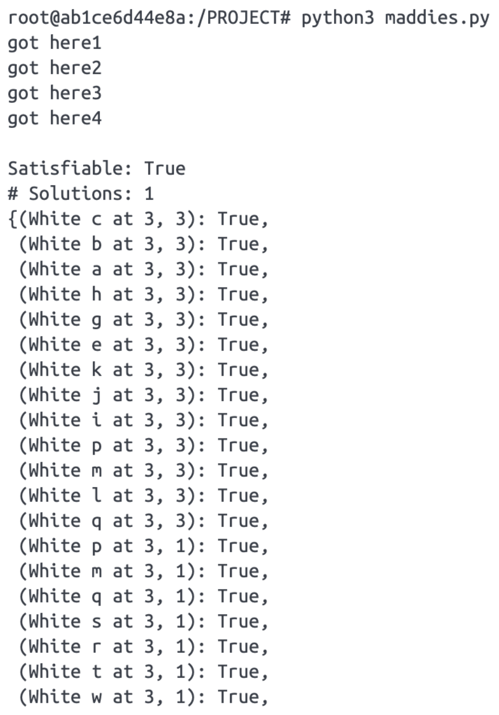
Running .introspect() on our encoding, it was observable that the assignment of all propositions as true or false was entirely the result of Python ”if” statements that determined if some letter, at some index x, y met some conditions (i.e. is the letter in the solution, what is the colour of the tile at the given index,etc.) prior to giving it a constraint, rather than through the use of logic symbols (&, |, >>)to establish the same conditional relationships.

Better Implementation of Logic

The conclusions we made of the implementation above led us to completely rework the logic and constraints that determined the truth of some proposition. In doing so, we once again changed our propositions, as seen in Figure 13.

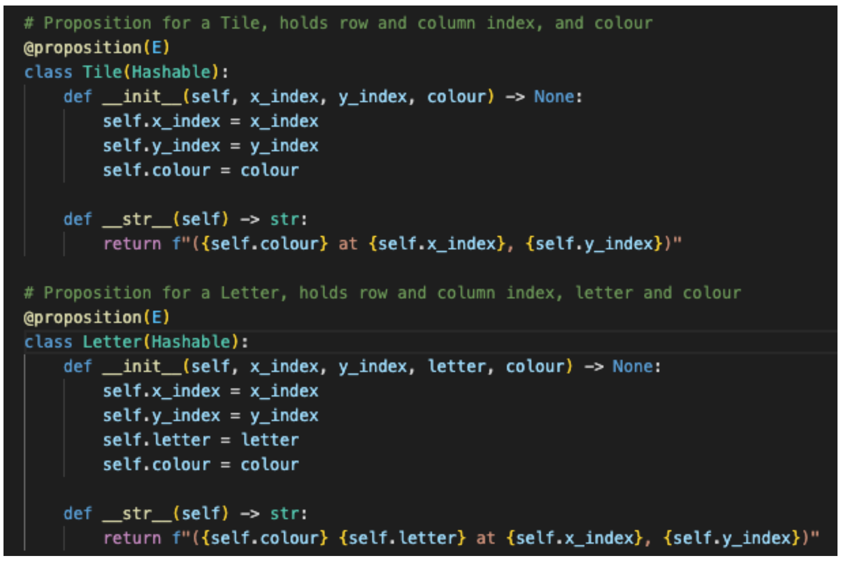
We built on the single proposition Tile$_{(x,y,l,c)}$, in favour of two propositions Tile$_{(x,y,c)}$, which represents the colour of some tile in the empty board layout at x, y and Letter$_{(x,y,l,c)}$ which represents exactly what Tile$_{(x,y,l,c)}$ did previously; some letter l assigned to some index x, y with some colour c.

Having learned from previous implementations, we knew it would be necessary to establish some kind of ”base case” - the absolute truth of the solution letters in the final row. The constraint to achieve such is seen in Figure 14.



```
root@ab1ce6d44e8a:/PROJECT# python3 maddies.py
got here1
got here2
got here3
got here4

Satisfiable: True
# Solutions: 1
{(White c at 3, 3): True,
 (White b at 3, 3): True,
 (White a at 3, 3): True,
 (White h at 3, 3): True,
 (White g at 3, 3): True,
 (White e at 3, 3): True,
 (White k at 3, 3): True,
 (White j at 3, 3): True,
 (White i at 3, 3): True,
 (White p at 3, 3): True,
 (White m at 3, 3): True,
 (White l at 3, 3): True,
 (White q at 3, 3): True,
 (White p at 3, 1): True,
 (White m at 3, 1): True,
 (White q at 3, 1): True,
 (White s at 3, 1): True,
 (White r at 3, 1): True,
 (White t at 3, 1): True,
 (White w at 3, 1): True,
```

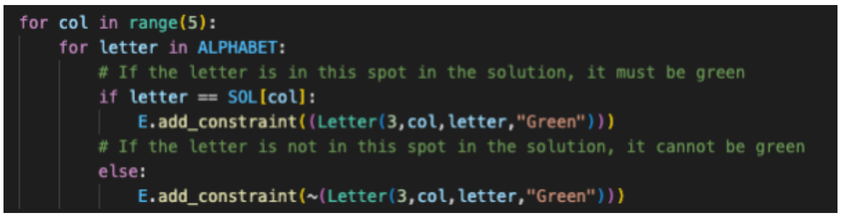
Figure 12: Sample of Letter Proposition Assignments



```python
# Proposition for a Tile, holds row and column index, and colour
@proposition(E)
class Tile(Hashable):
    def __init__(self, x_index, y_index, colour) -> None:
        self.x_index = x_index
        self.y_index = y_index
        self.colour = colour

    def __str__(self) -> str:
        return f"({self.colour} at {self.x_index}, {self.y_index})"

# Proposition for a Letter, holds row and column index, letter and colour
@proposition(E)
class Letter(Hashable):
    def __init__(self, x_index, y_index, letter, colour) -> None:
        self.x_index = x_index
        self.y_index = y_index
        self.letter = letter
        self.colour = colour

    def __str__(self) -> str:
        return f"({self.colour} {self.letter} at {self.x_index}, {self.y_index})"
```

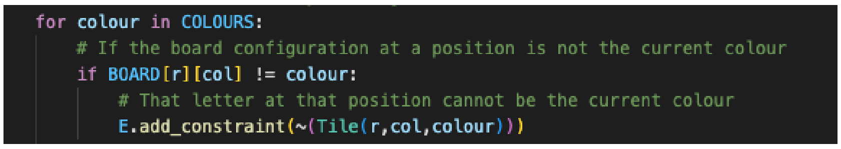
Figure 13: Final Propositions

We then create tile propositions by reading colour of board input, such that the model has some understanding of which colours are located where on the board layout, seen in Figure 15.



```
for col in range(5):
    for letter in ALPHABET:
        # If the letter is in this spot in the solution, it must be green
        if letter == SOL[col]:
            E.add_constraint((Letter(3,col,letter,"Green")))
        # If the letter is not in this spot in the solution, it cannot be green
        else:
            E.add_constraint(~(Letter(3,col,letter,"Green")))
```

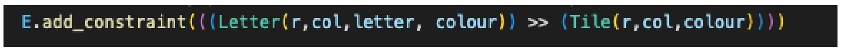
Figure 14: Constraining Truth of Row 3 Letters



```
for colour in COLOURS:
    # If the board configuration at a position is not the current colour
    if BOARD[r][col] != colour:
        # That letter at that position cannot be the current colour
        E.add_constraint(~(Tile(r,col,colour)))
```

Figure 15: Establishing Colour Locations on Board

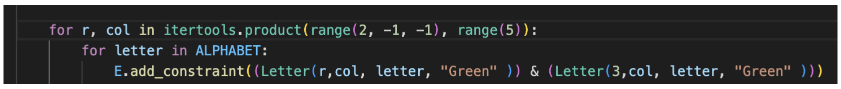
We then constrain that a letter of some colour c cannot be true unless the tile at that index is also of the same colour, seen in Figure 16. Creating a condition that in our previous implementation that had been established with a Python "if" statement, now done with logic.



```
E.add_constraint(((Letter(r,col,letter, colour)) >> (Tile(r,col,colour))))
```

Figure 16: Letter Propositon and Tile Proposition at Same Index Must Have Same Colour

Initially, we tried to develop our constraints using logical conjunctions, as can be seen in Figure 17, which did not produce the desired results. We realized that the use of a conjunction meant that when one proposition was False, the other became False by association. For example, suppose we were to say that for some proposition with a letter l and "Green" colour at some index x, y the proposition must also be true for letter l, colour "Green" in $x, 3$. This logic falls apart because it needs to be the case that letter l, colour "Green" in $x, 3$ can be true regardless of the truth of letter l and "Green" colour at some other index x, y, hence we transitioned to implications. As such, the right side of the implication to can be True even if the left side is False, but also ensures where the left side was True (suppose some letter is green at some index), so too must be the right side (letter must also be in the solution). True can only imply True, but False can imply anything! Early conjunction approaches are seen in Figure 15, and the lackluster results of that constraint are available in Figure 16. Improved constraints with the use of implications and the associated solutions are observed in Figures 17 and 18, respectively.

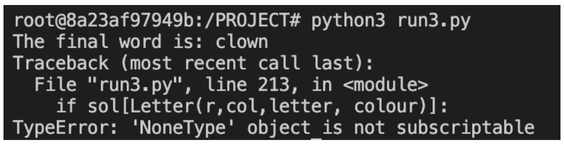


```
for r, col in itertools.product(range(2, -1, -1), range(5)):
    for letter in ALPHABET:
        E.add_constraint((Letter(r,col, letter, "Green" )) & (Letter(3,col, letter, "Green" )))
```

Figure 17: Constraint With Conjunction

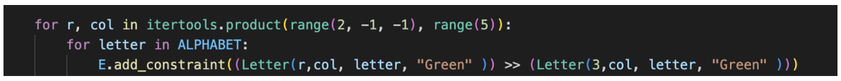
Two Build Theories

A problem we encountered in the early stages of our implementation was the struggle to iterate through the 26 letters in the alphabet, for each 3 colours, for all five x indices in a row, for each



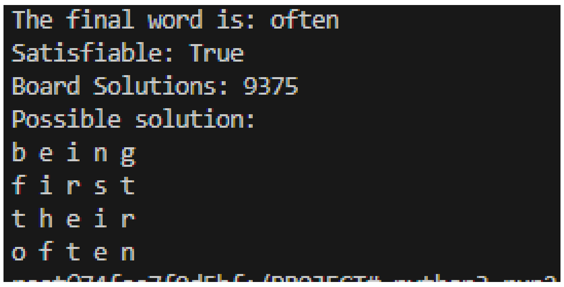
```
root@8a23af97949b:/PROJECT# python3 run3.py
The final word is: clown
Traceback (most recent call last):
  File "run3.py", line 213, in <module>
    if sol[Letter(r,col,letter, colour)]:
TypeError: 'NoneType' object is not subscriptable
```

Figure 18: Failed Solver From Constraint With Conjunction



```
for r, col in itertools.product(range(2, -1, -1), range(5)):
    for letter in ALPHABET:
        E.add_constraint((Letter(r,col, letter, "Green" )) >> (Letter(3,col, letter, "Green" )))
```

Figure 19: Constraint with Implication



```
The final word is: often
Satisfiable: True
Board Solutions: 9375
Possible solution:
b e i n g
f i r s t
t h e i r
o f t e n
```

Figure 20: Model Solutions

3 rows, to determine whether or not that permutation of five letters, each with some colour, satisfied a valid row solution. Our first such implementation is available in Figure 21. This implementation consistently killed our program, causing a run-time error.

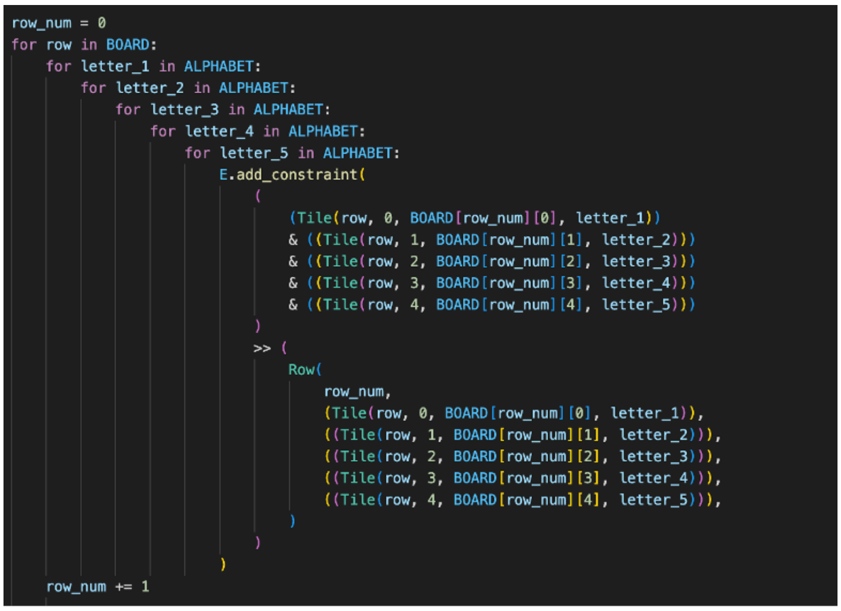
As such, we knew that we were going to have to reduce the number of permutations we iterated through in checking for valid row solutions, and the way to do that was to capture all potentially true letter propositions in some kind of data structure to be iterated over later. For example, for some green tile at index x, y, there is only ever going to be one satisfiable letter proposition, so there is no point in iterating through 26 letters at that index, when determining potential row solutions.

In our first implementation, capturing ”True” propositions at some index was easy as as result of our ”if” statements. However, after changing how our implementation in favour better use of logic, it was going to be impossible to store True propositions in an array structure, because the truth of a given proposition would not be known until after compilation.

To resolve this we split our build theory into two parts. The first build theory lays out all the constraint for the letter propositions, this is compiled and then all solutions of the first build theory are stored in a data structure (Seen in Figure 22) that is then passed to a second build theory to determine all valid rows and boards by our constraints.

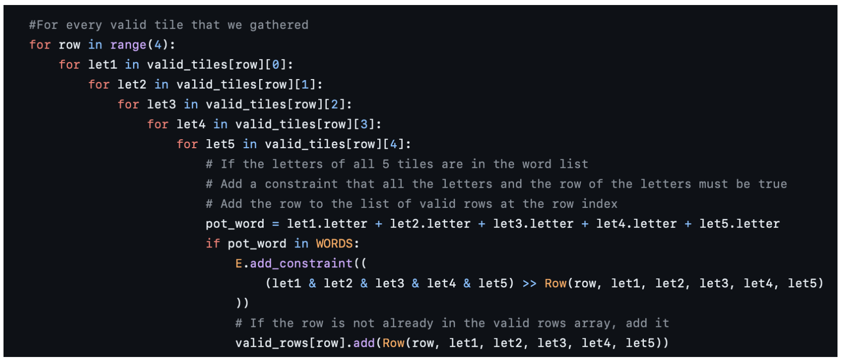
Final Output and Optimization

Finally, we optimize the readability of our solution output for the sake of error checking by colouring each letter in a row according to the empty board colouring. A sample of this output



```
row_num = 0
for row in BOARD:
    for letter_1 in ALPHABET:
        for letter_2 in ALPHABET:
            for letter_3 in ALPHABET:
                for letter_4 in ALPHABET:
                    for letter_5 in ALPHABET:
                        E.add_constraint(
                            (
                                (Tile(row, 0, BOARD[row_num][0], letter_1))
                                & ((Tile(row, 1, BOARD[row_num][1], letter_2)))
                                & ((Tile(row, 2, BOARD[row_num][2], letter_3)))
                                & ((Tile(row, 3, BOARD[row_num][3], letter_4)))
                                & ((Tile(row, 4, BOARD[row_num][4], letter_5)))
                            )
                            >> (
                                Row(
                                    row_num,
                                    (Tile(row, 0, BOARD[row_num][0], letter_1)),
                                    ((Tile(row, 1, BOARD[row_num][1], letter_2))),
                                    ((Tile(row, 2, BOARD[row_num][2], letter_3))),
                                    ((Tile(row, 3, BOARD[row_num][3], letter_4))),
                                    ((Tile(row, 4, BOARD[row_num][4], letter_5))),
                                )
                            )
                        )
    row_num += 1
```

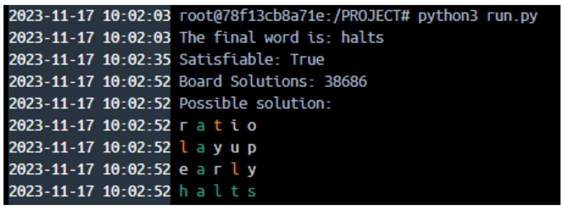
Figure 21: Initial Row Creation Loops - Killed Solver



```
#For every valid tile that we gathered
for row in range(4):
    for let1 in valid_tiles[row][0]:
        for let2 in valid_tiles[row][1]:
            for let3 in valid_tiles[row][2]:
                for let4 in valid_tiles[row][3]:
                    for let5 in valid_tiles[row][4]:
                        # If the letters of all 5 tiles are in the word list
                        # Add a constraint that all the letters and the row of the letters must be true
                        # Add the row to the list of valid rows at the row index
                        pot_word = let1.letter + let2.letter + let3.letter + let4.letter + let5.letter
                        if pot_word in WORDS:
                            E.add_constraint((
                                (let1 & let2 & let3 & let4 & let5) >> Row(row, let1, let2, let3, let4, let5)
                            ))
                            # If the row is not already in the valid rows array, add it
                            valid_rows[row].add(Row(row, let1, let2, let3, let4, let5))
```

Figure 22: Updated Letter Iteration to Determine Rows

can be seen in Figure 23.



```
2023-11-17 10:02:03 root@78f13cb8a71e:/PROJECT# python3 run.py
2023-11-17 10:02:03 The final word is: halts
2023-11-17 10:02:35 Satisfiable: True
2023-11-17 10:02:52 Board Solutions: 38686
2023-11-17 10:02:52 Possible solution:
2023-11-17 10:02:52 r a t i o
2023-11-17 10:02:52 l a y u p
2023-11-17 10:02:52 e a r l y
2023-11-17 10:02:52 h a l t s
```

Figure 23: Coloured Solution Output

Thorough out the process of our development we also had to be mindful of the length of our five letter word list, which we shortened and expanded as we explored our model. In some cases, the word list was a few as 10 words and our model was returning as many as 10,0000 solutions, see Figure Y. We had to be mindful of the length of our wordlist, as it had a significant impact on the run-time of over solver. One of the major hurdles we overcame was runtime. When keeping track of valid tiles, rows, and boards, lots of duplicate propositions and constraints were being generated. To fix this issue, and massively optimize our project, we checked if a proposition was not already in the list before adding it. At first, a board and word that resulted in single-digit solutions would get 'Killed' by the docker container. After implementing these changes, it would compile in seconds. Even a board and word that resulted in over a thousand solutions took less than a minute to compile.

We were still not satisfied with the time it took to to compile, so we changed all of the inner valid lists to sets, which made the contains check redundant. This removed thousands of comparisons which optimized our code even further. Maximum length word list (3834) and a board with many solutions (1000) takes too long to compute Limit the word list, right now it's at 2000 words and has a reasonable runtime.

First Order Extension

Domain of Discourse

i, x, y	row indices	$1 - 4$
j, x, y	column indices	$1 - 5$
l, k	letters	$a - z$
c	colours	green (G), yellow (Y), white (W)
w	words	words in our word list
p	permutations of letters	$a - z$

Predicates:

$T(i, j, l, c)$	a tile at row index i, column index j, with letter l, of colour c
$R(i, p)$	a row at index i, with a permutation of letters p
$SolR(i, w)$	a solution row at index i, with word w
$Board(w1, w2, w3)$	a solved board with words $1 - 3$, in their respective rows.

Formulas:

Tiles with the same row and same letter, in different columns do not exist, i.e., no duplicate letters in a row:
$\forall i.\forall x.\forall j.\forall y.\forall c.\exists l.\exists k.\neg(T(i,j,l) \land T(x,y,k) \rightarrow (i = x) \land (j \neq y) \land (l = k))$

All green tiles in the same column have the same letter:
$\forall i.\forall x.\forall y.\forall l.\forall k.((x = y) \land T(i,x,l,G) \land T(i,y,k,G) \rightarrow (l = k))$

All white tiles have different letters than green tiles:
$\forall i.\forall j.\forall l.\forall k.(T(i,j,l,W) \land T(i,j,k,G) \rightarrow (l \neq k))$

Yellow and green tiles have the same letters but at different column indices:
$\forall i.\forall j.\forall l.\forall k.\exists x.\exists k.(T(i,j,l,Y) \land T(i,x,k,G) \rightarrow (l = k) \land (j \neq x))$

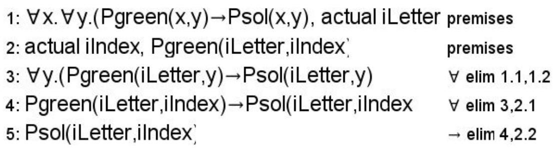
5 tiles with their respective letters and colors form a row with a
permutation of letters:
$\forall i.\exists l1.\exists l2.\exists l3.\exists l4.\exists l5.\exists c1.\exists c2.\exists c3.\exists c4.\exists c5.\exists p.(T(i,j,l1,c1) \land T(i,j,l2,c2) \land T(i,j,l3,c3)$
$\land T(i,j,l4,c4) \land T(i,j,l5,c5) \rightarrow R(i,p))$

For a row with a permutation of letters, if the permutation of letters is a word in our word list, it implies that the row is a solution row:
$\forall i.\forall p.\exists w.((p = w) \land R(i,p) \rightarrow SolR(i,w))$

If each row has a solution word, the board is a valid solution:
$\exists w1.\exists w2.\exists w3.(SolR(1,w1) \land SolR(2,w2) \land SolR(3,w3) \rightarrow Board(w1,w2,w3))$

Jape Proofs

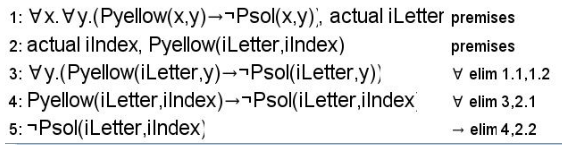
A green tile with letter x in column index y means there is a solution tile with the same letter and same column index :



```
1: ∀x.∀y.(Pgreen(x,y)→Psol(x,y), actual iLetter premises
2: actual iIndex, Pgreen(iLetter,iIndex)          premises
3: ∀y.(Pgreen(iLetter,y)→Psol(iLetter,y)          ∀ elim 1.1,1.2
4: Pgreen(iLetter,iIndex)→Psol(iLetter,iIndex     ∀ elim 3,2.1
5: Psol(iLetter,iIndex)                           → elim 4,2.2
```

Figure 24: Green Tile implies same Solution Tile

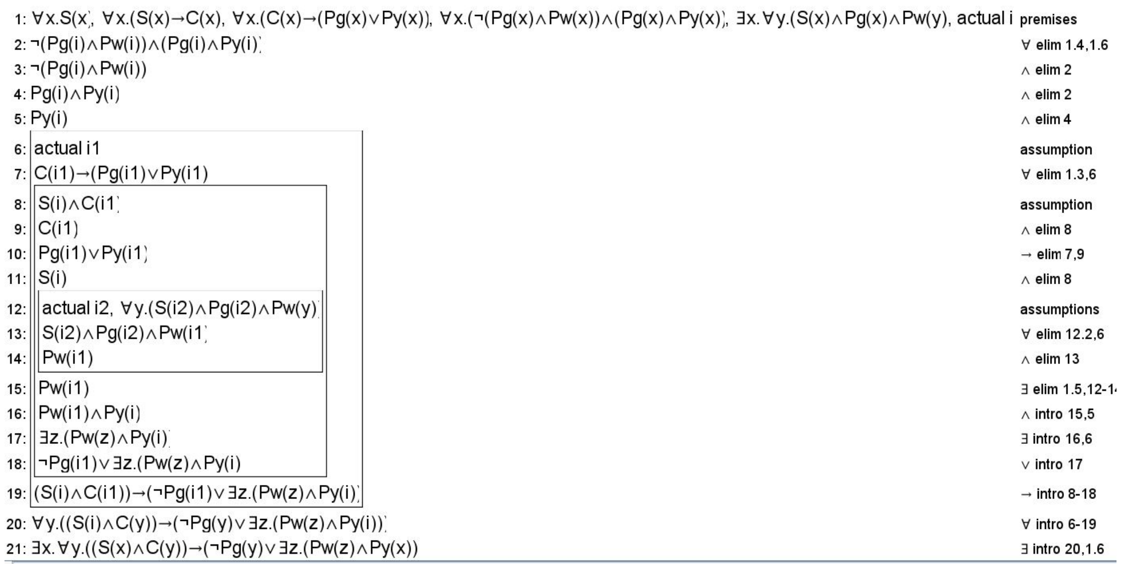
A yellow tile with the letter x in the column index y means that there is not a solution tile with the same letter and at that index:



```
1: ∀x.∀y.(Pyellow(x,y)→¬Psol(x,y), actual iLetter   premises
2: actual iIndex, Pyellow(iLetter,iIndex)           premises
3: ∀y.(Pyellow(iLetter,y)→¬Psol(iLetter,y)          ∀ elim 1.1,1.2
4: Pyellow(iLetter,iIndex)→¬Psol(iLetter,iIndex     ∀ elim 3,2.1
5: ¬Psol(iLetter,iIndex                             → elim 4,2.2
```

Figure 25: Yellow Tile implies different Solution Tile

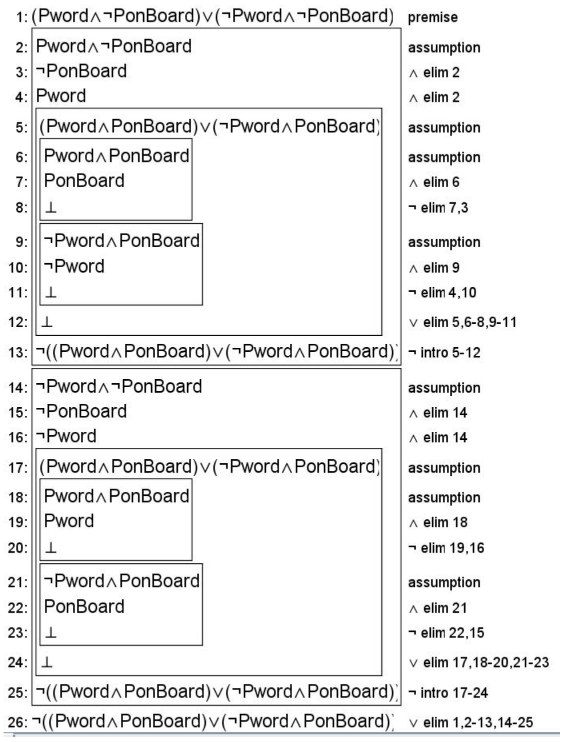
All tiles have a colour. A tile can be green white or yellow, letters will be green and yellow, or white, if the letter is green and yellow the tiles are at different spots on the board, white tiles have different locations then the green and yellow letters on the board.



```
1: ∀x.S(x, ∀x.(S(x)→C(x), ∀x.(C(x)→(Pg(x)∨Py(x)), ∀x.(¬(Pg(x)∧Pw(x))∧(Pg(x)∧Py(x), ∃x.∀y.(S(x)∧Pg(x)∧Pw(y), actual i   premises
2: ¬(Pg(i)∧Pw(i))∧(Pg(i)∧Py(i)                          ∀ elim 1.4,1.6
3: ¬(Pg(i)∧Pw(i))                                        ∧ elim 2
4: Pg(i)∧Py(i)                                           ∧ elim 2
5: Py(i)                                                 ∧ elim 4
6:  | actual i1                                          assumption
7:  | C(i1)→(Pg(i1)∨Py(i1)                               ∀ elim 1.3,6
8:  | | S(i)∧C(i1                                        assumption
9:  | | C(i1)                                            ∧ elim 8
10: | | Pg(i1)∨Py(i1                                     → elim 7,9
11: | | S(i)                                             ∧ elim 8
12: | | | actual i2, ∀y.(S(i2)∧Pg(i2)∧Pw(y)              assumptions
13: | | | S(i2)∧Pg(i2)∧Pw(i1                            ∀ elim 12.2,6
14: | | | Pw(i1)                                         ∧ elim 13
15: | | Pw(i1)                                           ∃ elim 1.5,12-1
16: | | Pw(i1)∧Py(i)                                     ∧ intro 15,5
17: | | ∃z.(Pw(z)∧Py(i)                                  ∃ intro 16,6
18: | | ¬Pg(i1)∨∃z.(Pw(z)∧Py(i)                          ∨ intro 17
19: | (S(i)∧C(i1))→(¬Pg(i1)∨∃z.(Pw(z)∧Py(i)             → intro 8-18
20: ∀y.((S(i)∧C(y))→(¬Pg(y)∨∃z.(Pw(z)∧Py(i))            ∀ intro 6-19
21: ∃x.∀y.((S(x)∧C(y))→(¬Pg(y)∨∃z.(Pw(z)∧Py(x))          ∃ intro 20,1.6
```

Figure 26: Rules for the colouring

A word that is not on the board or a permutation that is not a word that is also not on the board implies the letters in the word or permutation will not show up in the solution:



```
1:  (Pword∧¬PonBoard)∨(¬Pword∧¬PonBoard)        premise
2:  Pword∧¬PonBoard                             assumption
3:  ¬PonBoard                                   ∧ elim 2
4:  Pword                                       ∧ elim 2
5:    (Pword∧PonBoard)∨(¬Pword∧PonBoard)        assumption
6:      Pword∧PonBoard                          assumption
7:      PonBoard                                ∧ elim 6
8:      ⊥                                       ¬ elim 7,3
9:      ¬Pword∧PonBoard                         assumption
10:     ¬Pword                                  ∧ elim 9
11:     ⊥                                       ¬ elim 4,10
12:   ⊥                                         ∨ elim 5,6-8,9-11
13: ¬((Pword∧PonBoard)∨(¬Pword∧PonBoard))       ¬ intro 5-12
14: ¬Pword∧¬PonBoard                            assumption
15: ¬PonBoard                                   ∧ elim 14
16: ¬Pword                                      ∧ elim 14
17:   (Pword∧PonBoard)∨(¬Pword∧PonBoard)        assumption
18:     Pword∧PonBoard                          assumption
19:     Pword                                   ∧ elim 18
20:     ⊥                                       ¬ elim 19,16
21:     ¬Pword∧PonBoard                         assumption
22:     PonBoard                                ∧ elim 21
23:     ⊥                                       ¬ elim 22,15
24:   ⊥                                         ∨ elim 17,18-20,21-23
25: ¬((Pword∧PonBoard)∨(¬Pword∧PonBoard))       ¬ intro 17-24
26: ¬((Pword∧PonBoard)∨(¬Pword∧PonBoard))       ∨ elim 1,2-13,14-25
```

Figure 27: Invalid permutations are not on the final board