Rush Hour

Jacob Phillipson (21JZSP)
Joel Mills (21JAM71)
Alexander Schmelzer (21AES30)
Ethan Goldberg (21EHG2)

Group 18

Abstract

Rush Hour is a classic logic game that involves sliding vehicles across a grid to enable the red vehicle's escape

This project aims to create a detailed simulation of Rush Hour using a model that represents the state of each cell in the game grid. Cells will be numbered in the form (row, column), counting from the top left, which is cell (0,0).

Our model will correspond to the states of every cell. Every cell will have a true proposition representing the vehicle type that is in that cell. Every cell will also have propositions that will be true if it is the lead cell in a car that can move (For example, the rightmost cell in a car that can move right).

We will use our model to simulate the game of Rush Hour according to its rules by figuring out which pieces can move. From that base, we can then build a playable game. We can also run this process in reverse to generate playable boards.

This project aims to model and simulate the game 'Rush Hour' using logical propositions and constraints, providing an interactive and educational tool for understanding logic as applied to simple game theory.

Propositions

The following propositions form the foundation of our Rush Hour model, each representing a unique state or rule within the game.

For each proposition, their i, j values represent which cell on the board that proposition is referring to.
Following is a list of all propositions:

- E_{ij}: If this is true, the cell is empty
- Using coordinates (k, j) where k is some constant between 0 and 5 that cannot change.
  - Red_{kj}: If this is true, the cell contains the red car
  - $H2_{kj}$: If this is true, the cell contains a horizontal car of length 2
  - $H3_{kj}$: If this is true, the cell contains a horizontal car of length 3
- Using coordinates (i, k) where k is some constant between 0 and 5 that cannot change.
  - $V2_{ik}$: If this is true, the cell contains a vertical car of length 2
  - $V3_{ik}$: If this is true, the cell contains a vertical car of length 3
- Using coordinates (i, j)

  - CMR_{ij}: If this is true, cell (i,j) is the rightmost cell of a car that can move right.
  - CML_{ij}: If this is true, cell (i,j) is the leftmost cell of a car that can move left.
  - CMD_{ij}: If this is true, cell (i,j) is the lowest cell of a car that can move down.
  - CMU_{ij}: If this is true, cell (i,j) is the highest cell of a car that can move up.

Constraints

The following constraints are necessary to ensure the game's logical consistency and playability.

Movement Constraints (Implemented through Bauhaus)

- A cell must have exactly one value be true at any time:

$$V2_{ij} \leftrightarrow \neg(V3_{ij} \vee H2_{ij} \vee H3_{ij} \vee E_{ij} \vee R_{ij})...$$

- The red vehicle must be on the third row of the grid, aligned with the exit

$$R_{2j}$$

- We start with some initial configuration, where every cell must be assigned:

$$E_{00} \wedge H2_{01} \wedge H2_{02} \wedge H3_{03} \wedge H3_{04} \wedge H3_{05} \wedge \dots$$

- When one part of a car moves, the other parts of the car must move the same amount in the same direction

$$H3_{10} \wedge H3_{11} \wedge H3_{12} \text{ becomes } H3_{11} \wedge H3_{12} \wedge H3_{13}$$

Game Constraints (Implemented through Python)

- The red car's dimensions must be 2 cells in width and 1 cell in height.
- The game board must have dimensions of 6x6

$$\text{range of } i, j = [0, 5]$$

- Cars have a fixed size
- No vehicles moving in the same direction as the red car can be in the same row as the red car.

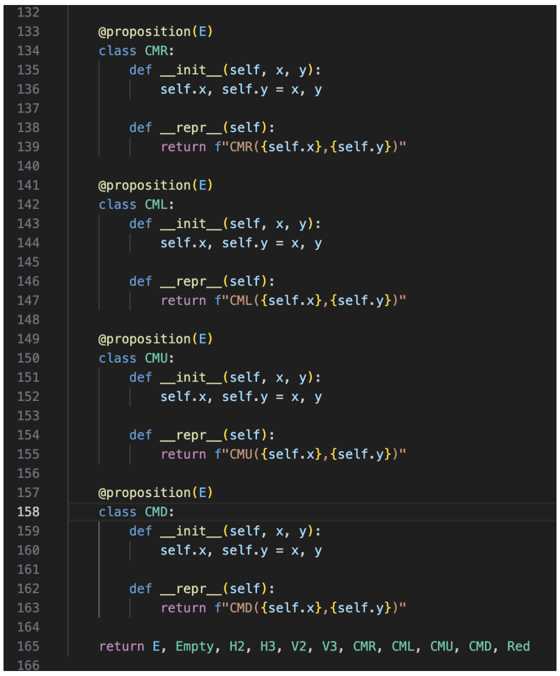
$$\neg(H3_{2j} \vee H2_{2j})$$

- The game is complete when the red car reaches the exit:
  - The exit must be to the right of cell (2, 5)
  - $Red_{25} \rightarrow W$

Model Exploration

We encountered many different logical challenges throughout the building of our model, stemming from both discoveries about how the model functioned and challenges properly implementing the logic in a way that represented what we were trying to implement.
The first things we noticed related to our original propositions and constraints. We found that they did not represent the model in the best way possible, so we modified them a couple of times throughout the project:

- We found that "Vehicles cannot overlap or go through other vehicles on the board" and "A cell must have exactly one value be true at any time" were the same constraint, with the latter being more in-depth, so we removed the first constraint.
- We removed the following as it was added in the propositions:
  - "k is a constant value that remains the same for all V-type cars, regardless of the model, as the column cannot change. In simple terms, all V-type cars can be uniquely identified by their row value, as their column values are constant."
  - "k is a constant value that remains the same for all H-type cars, regardless of the model, as the row cannot change. In simple terms, all H-type cars can be uniquely identified by their column value, as their row values are constant."
- We also refined our propositions for greater accuracy. For example, we added propositions representing a car's ability to move, which helped in depicting the dynamic nature of our game board. This addition proved crucial in simulating realistic game movements.

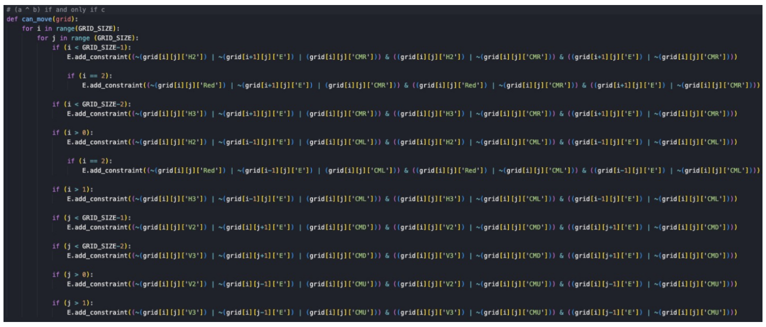


```
    @proposition(E)
    class CMR:
        def __init__(self, x, y):
            self.x, self.y = x, y

        def __repr__(self):
            return f"CMR({self.x},{self.y})"

    @proposition(E)
    class CML:
        def __init__(self, x, y):
            self.x, self.y = x, y

        def __repr__(self):
            return f"CML({self.x},{self.y})"

    @proposition(E)
    class CMU:
        def __init__(self, x, y):
            self.x, self.y = x, y

        def __repr__(self):
            return f"CMU({self.x},{self.y})"

    @proposition(E)
    class CMD:
        def __init__(self, x, y):
            self.x, self.y = x, y

        def __repr__(self):
            return f"CMD({self.x},{self.y})"

    return E, Empty, H2, H3, V2, V3, CMR, CML, CMU, CMD, Red
```

- We were struggling to figure out how to simulate a move. So we added propositions to represent if a lead car cell can move and in which direction. We can now loop through the cells and see what pieces can move and what the result will be.
- Originally, we had extra proposition classes for previous and next cell values. We realized we did not need to specify between the previous and next propositions as we already have a previous and next grid; we can use the same proposition classes for each.
- Based on how our check valid moves function operates, there is no need to separately assert that cars cannot move antithesis to their definition. (images of the final 4 move functions can be seen below)

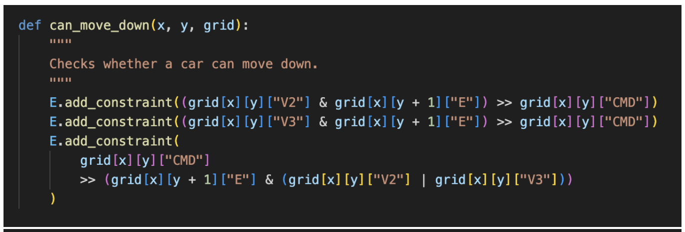
The functions around how the cars moved in the context of the game took us a long time to properly implement. This section presented an interesting test of our understanding of logic and the implementation of the Bauhaus library and its logical formulae. We went back and forth on the way the functions were written until we found something that represented what we wanted:

- For a long time, they were failing to properly register as constraints which resulted in our move propositions not functioning properly. At first, we assumed that the propositions were not formatted properly for the solver, so we performed implication elimination and deMorgan's on all the formulae to put them in NNF form.

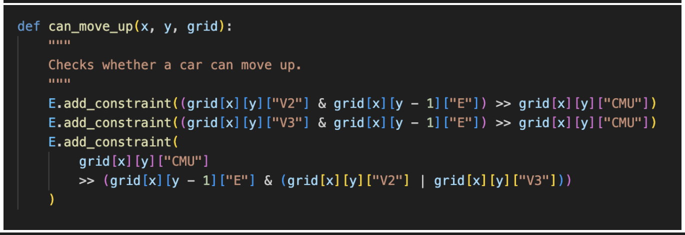


```python
# (a ^ b) if and only if c
def can_move(grid):
    for i in range(GRID_SIZE):
        for j in range (GRID_SIZE):
            if (i < GRID_SIZE-1):
                E.add_constraint((~(grid[i][j]['H2']) | ~(grid[i+1][j]['E']) | (grid[i][j]['CMR'])) & ((grid[i][j]['H2']) | ~(grid[i][j]['CMR'])) & ((grid[i+1][j]['E']) | ~(grid[i][j]['CMR'])))

                if (i == 2):
                    E.add_constraint((~(grid[i][j]['Red']) | ~(grid[i+1][j]['E']) | (grid[i][j]['CMR'])) & ((grid[i][j]['Red']) | ~(grid[i][j]['CMR'])) & ((grid[i+1][j]['E']) | ~(grid[i][j]['CMR'])))

            if (i < GRID_SIZE-2):
                E.add_constraint((~(grid[i][j]['H3']) | ~(grid[i+1][j]['E']) | (grid[i][j]['CMR'])) & ((grid[i][j]['H3']) | ~(grid[i][j]['CMR'])) & ((grid[i+1][j]['E']) | ~(grid[i][j]['CMR'])))

            if (i > 0):
                E.add_constraint((~(grid[i][j]['H2']) | ~(grid[i-1][j]['E']) | (grid[i][j]['CML'])) & ((grid[i][j]['H2']) | ~(grid[i][j]['CML'])) & ((grid[i-1][j]['E']) | ~(grid[i][j]['CML'])))

                if (i == 2):
                    E.add_constraint((~(grid[i][j]['Red']) | ~(grid[i-1][j]['E']) | (grid[i][j]['CML'])) & ((grid[i][j]['Red']) | ~(grid[i][j]['CML'])) & ((grid[i-1][j]['E']) | ~(grid[i][j]['CML'])))

            if (i > 1):
                E.add_constraint((~(grid[i][j]['H3']) | ~(grid[i-1][j]['E']) | (grid[i][j]['CML'])) & ((grid[i][j]['H3']) | ~(grid[i][j]['CML'])) & ((grid[i-1][j]['E']) | ~(grid[i][j]['CML'])))

            if (j < GRID_SIZE-1):
                E.add_constraint((~(grid[i][j]['V2']) | ~(grid[i][j+1]['E']) | (grid[i][j]['CMD'])) & ((grid[i][j]['V2']) | ~(grid[i][j]['CMD'])) & ((grid[i][j+1]['E']) | ~(grid[i][j]['CMD'])))

            if (j < GRID_SIZE-2):
                E.add_constraint((~(grid[i][j]['V3']) | ~(grid[i][j+1]['E']) | (grid[i][j]['CMD'])) & ((grid[i][j]['V3']) | ~(grid[i][j]['CMD'])) & ((grid[i][j+1]['E']) | ~(grid[i][j]['CMD'])))

            if (j > 0):
                E.add_constraint((~(grid[i][j]['V2']) | ~(grid[i][j-1]['E']) | (grid[i][j]['CMU'])) & ((grid[i][j]['V2']) | ~(grid[i][j]['CMU'])) & ((grid[i][j-1]['E']) | ~(grid[i][j]['CMU'])))

            if (j > 1):
                E.add_constraint((~(grid[i][j]['V3']) | ~(grid[i][j-1]['E']) | (grid[i][j]['CMU'])) & ((grid[i][j]['V3']) | ~(grid[i][j]['CMU'])) & ((grid[i][j-1]['E']) | ~(grid[i][j]['CMU'])))
```

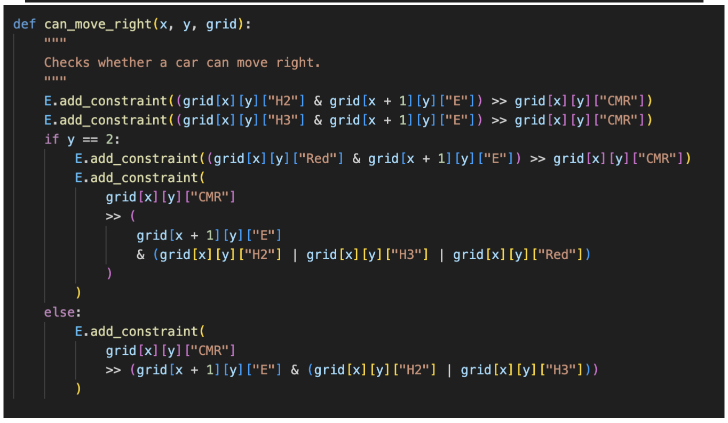
- We then discovered that this was not the reason the logic was failing. The constraints were registering; we just had a different logical error. The real breakthrough came when we realized the need for bidirectional implications in our constraints, ensuring that each logical relation was accurately represented in both directions.



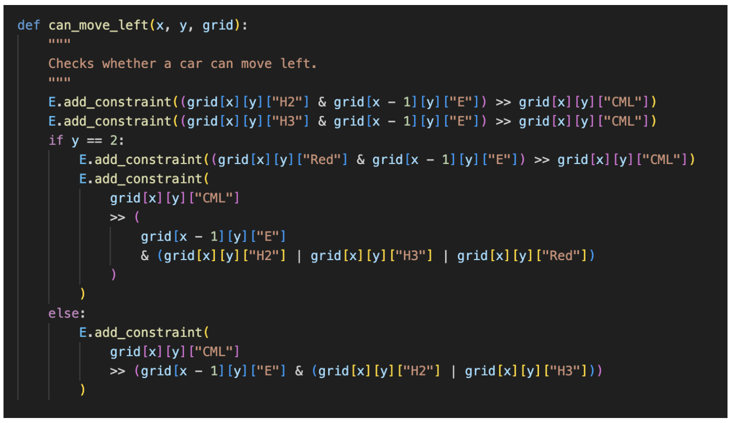
```python
def can_move_down(x, y, grid):
    """
    Checks whether a car can move down.
    """
    E.add_constraint((grid[x][y]["V2"] & grid[x][y + 1]["E"]) >> grid[x][y]["CMD"])
    E.add_constraint((grid[x][y]["V3"] & grid[x][y + 1]["E"]) >> grid[x][y]["CMD"])
    E.add_constraint(
        grid[x][y]["CMD"]
        >> (grid[x][y + 1]["E"] & (grid[x][y]["V2"] | grid[x][y]["V3"]))
    )
```



```python
def can_move_up(x, y, grid):
    """
    Checks whether a car can move up.
    """
    E.add_constraint((grid[x][y]["V2"] & grid[x][y - 1]["E"]) >> grid[x][y]["CMU"])
    E.add_constraint((grid[x][y]["V3"] & grid[x][y - 1]["E"]) >> grid[x][y]["CMU"])
    E.add_constraint(
        grid[x][y]["CMU"]
        >> (grid[x][y - 1]["E"] & (grid[x][y]["V2"] | grid[x][y]["V3"]))
    )
```



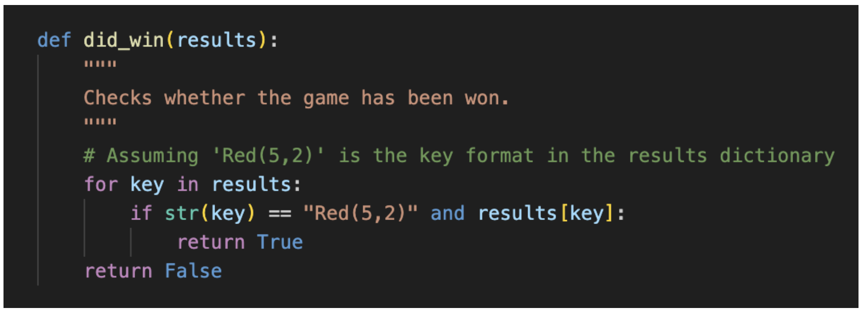
```python
def can_move_right(x, y, grid):
    """
    Checks whether a car can move right.
    """
    E.add_constraint((grid[x][y]["H2"] & grid[x + 1][y]["E"]) >> grid[x][y]["CMR"])
    E.add_constraint((grid[x][y]["H3"] & grid[x + 1][y]["E"]) >> grid[x][y]["CMR"])
    if y == 2:
        E.add_constraint((grid[x][y]["Red"] & grid[x + 1][y]["E"]) >> grid[x][y]["CMR"])
        E.add_constraint(
            grid[x][y]["CMR"]
            >> (
                grid[x + 1][y]["E"]
                & (grid[x][y]["H2"] | grid[x][y]["H3"] | grid[x][y]["Red"])
            )
        )
    else:
        E.add_constraint(
            grid[x][y]["CMR"]
            >> (grid[x + 1][y]["E"] & (grid[x][y]["H2"] | grid[x][y]["H3"]))
        )
```



```python
def can_move_left(x, y, grid):
    """
    Checks whether a car can move left.
    """
    E.add_constraint((grid[x][y]["H2"] & grid[x - 1][y]["E"]) >> grid[x][y]["CML"])
    E.add_constraint((grid[x][y]["H3"] & grid[x - 1][y]["E"]) >> grid[x][y]["CML"])
    if y == 2:
        E.add_constraint((grid[x][y]["Red"] & grid[x - 1][y]["E"]) >> grid[x][y]["CML"])
        E.add_constraint(
            grid[x][y]["CML"]
            >> (
                grid[x - 1][y]["E"]
                & (grid[x][y]["H2"] | grid[x][y]["H3"] | grid[x][y]["Red"])
            )
        )
    else:
        E.add_constraint(
            grid[x][y]["CML"]
            >> (grid[x - 1][y]["E"] & (grid[x][y]["H2"] | grid[x][y]["H3"]))
        )
```

Further enhancements included the development of a function to check for a winning state in the game, signifying the completion of a game round.:

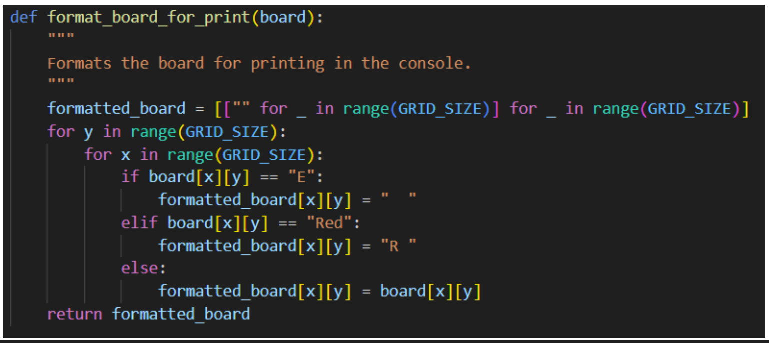
- We built a function to check if the current state of the board was a winning state in our model that would return true if the Red Car occupied its winning position, which would let us know the game was over.



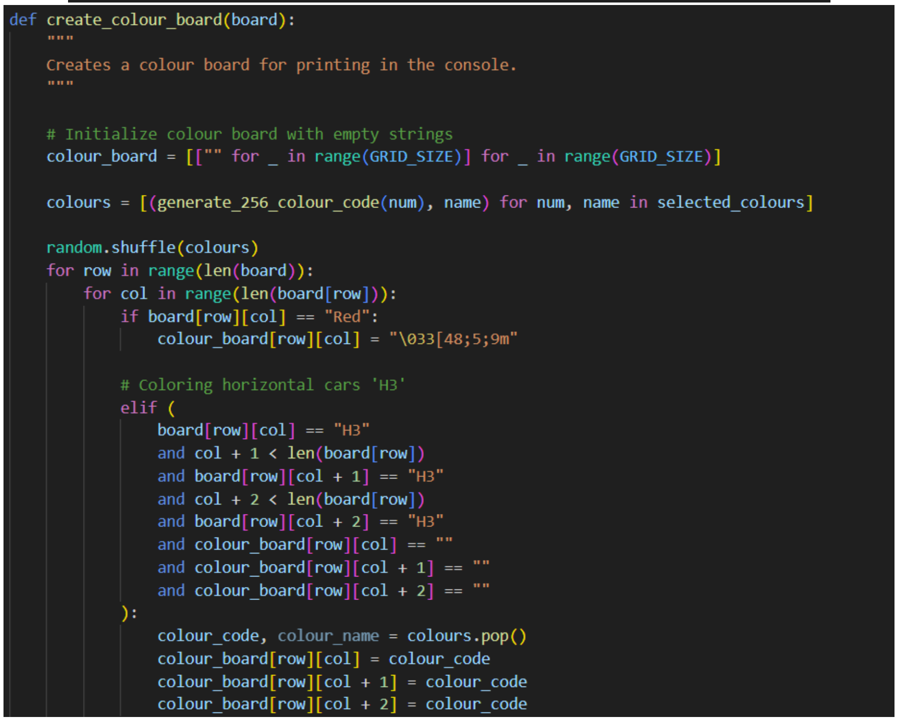
```python
def did_win(results):
    """
    Checks whether the game has been won.
    """
    # Assuming 'Red(5,2)' is the key format in the results dictionary
    for key in results:
        if str(key) == "Red(5,2)" and results[key]:
            return True
    return False
```

We felt an important part of the project was properly simulating a game of Rush Hour that was playable and functioned according to the rules of the game by using our propositions and constraints, so we set about to build a visual representation of a functioning playable game:

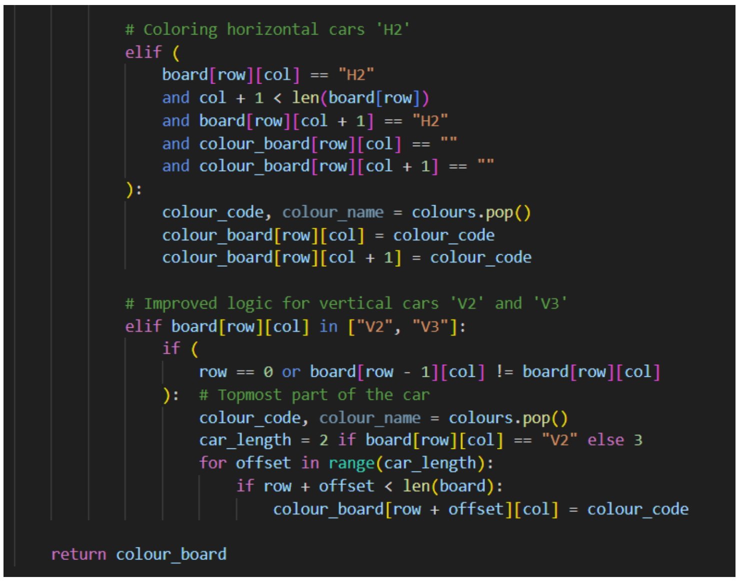
- First, we translated the necessary propositions into better visual representations. We also built colour functions to properly colour the board so all the cars had unique colours:



```python
def format_board_for_print(board):
    """
    Formats the board for printing in the console.
    """
    formatted_board = [["" for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]
    for y in range(GRID_SIZE):
        for x in range(GRID_SIZE):
            if board[x][y] == "E":
                formatted_board[x][y] = "  "
            elif board[x][y] == "Red":
                formatted_board[x][y] = "R "
            else:
                formatted_board[x][y] = board[x][y]
    return formatted_board
```

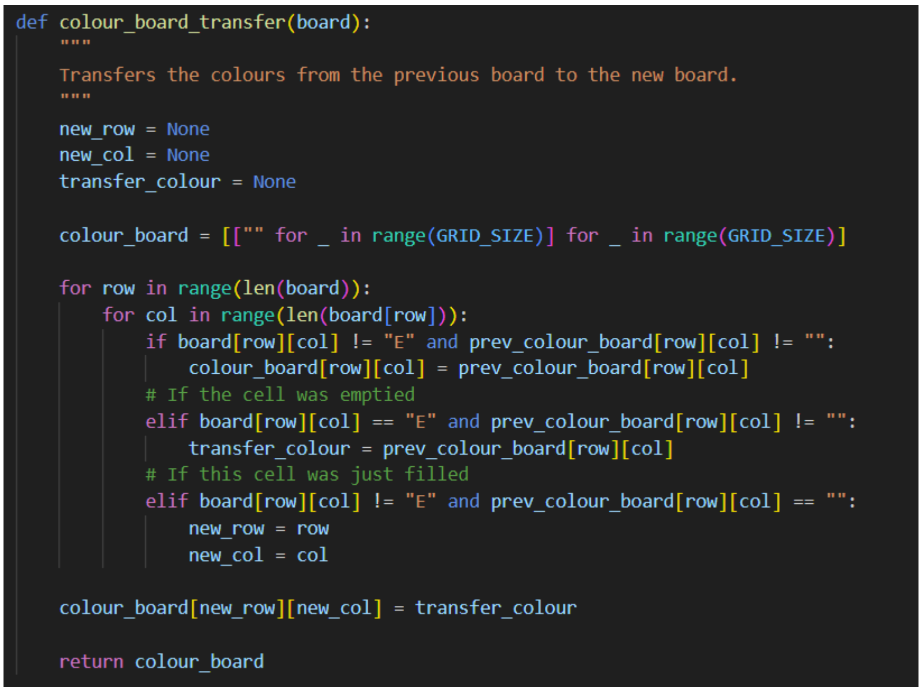


```python
def create_colour_board(board):
    """
    Creates a colour board for printing in the console.
    """

    # Initialize colour board with empty strings
    colour_board = [["" for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]

    colours = [(generate_256_colour_code(num), name) for num, name in selected_colours]

    random.shuffle(colours)
    for row in range(len(board)):
        for col in range(len(board[row])):
            if board[row][col] == "Red":
                colour_board[row][col] = "\033[48;5;9m"

            # Coloring horizontal cars 'H3'
            elif (
                board[row][col] == "H3"
                and col + 1 < len(board[row])
                and board[row][col + 1] == "H3"
                and col + 2 < len(board[row])
                and board[row][col + 2] == "H3"
                and colour_board[row][col] == ""
                and colour_board[row][col + 1] == ""
                and colour_board[row][col + 2] == ""
            ):
                colour_code, colour_name = colours.pop()
                colour_board[row][col] = colour_code
                colour_board[row][col + 1] = colour_code
                colour_board[row][col + 2] = colour_code
```



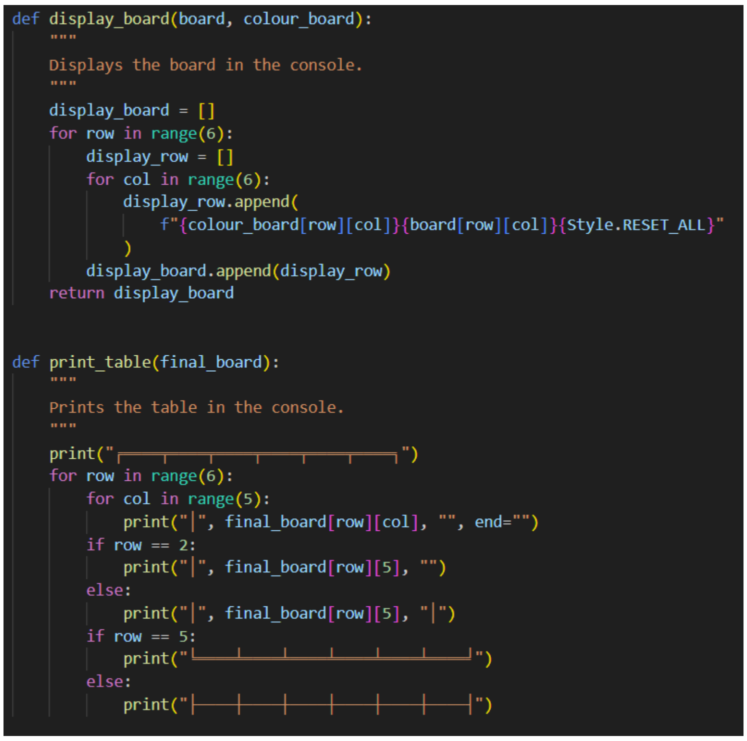
```
            # Coloring horizontal cars 'H2'
            elif (
                board[row][col] == "H2"
                and col + 1 < len(board[row])
                and board[row][col + 1] == "H2"
                and colour_board[row][col] == ""
                and colour_board[row][col + 1] == ""
            ):
                colour_code, colour_name = colours.pop()
                colour_board[row][col] = colour_code
                colour_board[row][col + 1] = colour_code

            # Improved logic for vertical cars 'V2' and 'V3'
            elif board[row][col] in ["V2", "V3"]:
                if (
                    row == 0 or board[row - 1][col] != board[row][col]
                ):  # Topmost part of the car
                    colour_code, colour_name = colours.pop()
                    car_length = 2 if board[row][col] == "V2" else 3
                    for offset in range(car_length):
                        if row + offset < len(board):
                            colour_board[row + offset][col] = colour_code

    return colour_board
```

- One part we found a little challenging to bug-fix was keeping the board colours consistent. Originally, the colours for the cars would change with every move, so we built a function that would ensure that cars stay the same colour for the entire game:



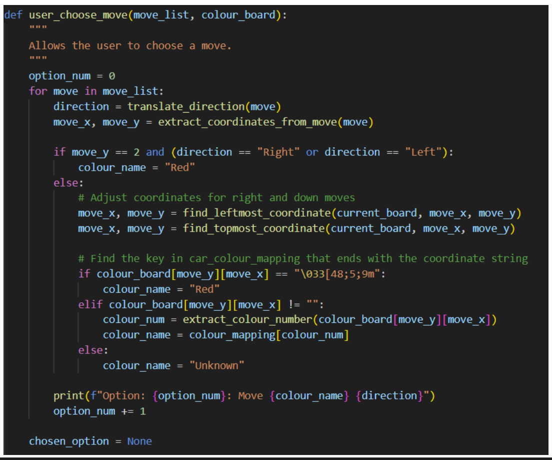
```
def colour_board_transfer(board):
    """
    Transfers the colours from the previous board to the new board.
    """
    new_row = None
    new_col = None
    transfer_colour = None

    colour_board = [["" for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]

    for row in range(len(board)):
        for col in range(len(board[row])):
            if board[row][col] != "E" and prev_colour_board[row][col] != "":
                colour_board[row][col] = prev_colour_board[row][col]
            # If the cell was emptied
            elif board[row][col] == "E" and prev_colour_board[row][col] != "":
                transfer_colour = prev_colour_board[row][col]
            # If this cell was just filled
            elif board[row][col] != "E" and prev_colour_board[row][col] == "":
                new_row = row
                new_col = col

    colour_board[new_row][new_col] = transfer_colour

    return colour_board
```

- We were then able to build a viewable board that a player could interact with:

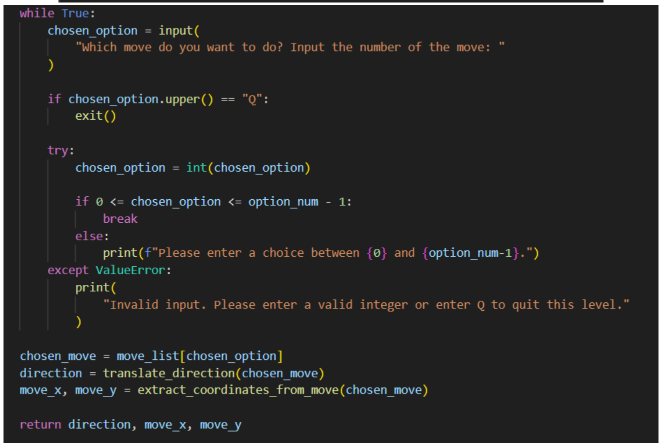


```python
def display_board(board, colour_board):
    """
    Displays the board in the console.
    """
    display_board = []
    for row in range(6):
        display_row = []
        for col in range(6):
            display_row.append(
                f"{colour_board[row][col]}{board[row][col]}{Style.RESET_ALL}"
            )
        display_board.append(display_row)
    return display_board


def print_table(final_board):
    """
    Prints the table in the console.
    """
    print("┌─────┬─────┬─────┬─────┬─────┬─────┐")
    for row in range(6):
        for col in range(5):
            print("│", final_board[row][col], "", end="")
        if row == 2:
            print("│", final_board[row][5], "")
        else:
            print("│", final_board[row][5], "│")
        if row == 5:
            print("└─────┴─────┴─────┴─────┴─────┴─────┘")
        else:
            print("├─────┼─────┼─────┼─────┼─────┼─────┤")
```

- With all of this setup, we were able to build a function that allowed a player to choose moves to play the game:

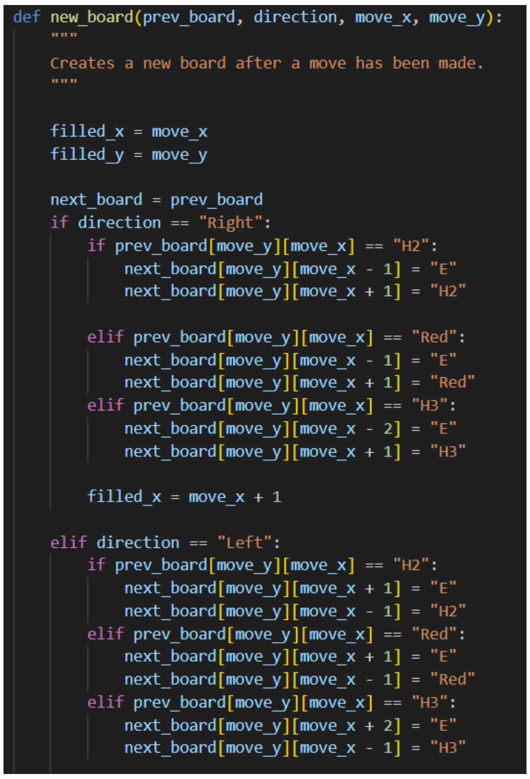


```python
def user_choose_move(move_list, colour_board):
    """
    Allows the user to choose a move.
    """
    option_num = 0
    for move in move_list:
        direction = translate_direction(move)
        move_x, move_y = extract_coordinates_from_move(move)

        if move_y == 2 and (direction == "Right" or direction == "Left"):
            colour_name = "Red"
        else:
            # Adjust coordinates for right and down moves
            move_x, move_y = find_leftmost_coordinate(current_board, move_x, move_y)
            move_x, move_y = find_topmost_coordinate(current_board, move_x, move_y)

            # Find the key in car_colour_mapping that ends with the coordinate string
            if colour_board[move_y][move_x] == "\033[48;5;9m":
                colour_name = "Red"
            elif colour_board[move_y][move_x] != "":
                colour_num = extract_colour_number(colour_board[move_y][move_x])
                colour_name = colour_mapping[colour_num]
            else:
                colour_name = "Unknown"

        print(f"Option: {option_num}: Move {colour_name} {direction}")
        option_num += 1

    chosen_option = None
```

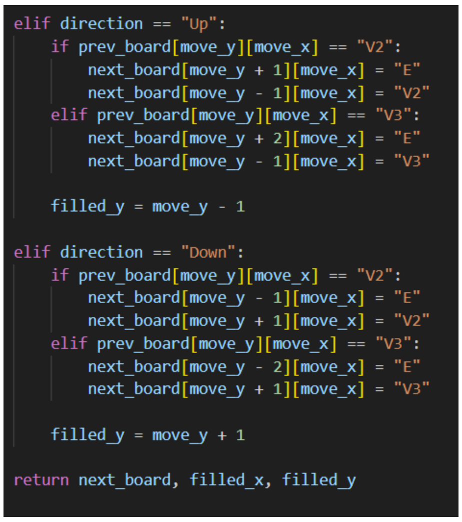


```python
    while True:
        chosen_option = input(
            "Which move do you want to do? Input the number of the move: "
        )

        if chosen_option.upper() == "Q":
            exit()

        try:
            chosen_option = int(chosen_option)

            if 0 <= chosen_option <= option_num - 1:
                break
            else:
                print(f"Please enter a choice between {0} and {option_num-1}.")
        except ValueError:
            print(
                "Invalid input. Please enter a valid integer or enter Q to quit this level."
            )

    chosen_move = move_list[chosen_option]
    direction = translate_direction(chosen_move)
    move_x, move_y = extract_coordinates_from_move(chosen_move)

    return direction, move_x, move_y
```

- From that point, all we had to do was update the current board to the one that included the move change, and it could repeat until the game was completed:



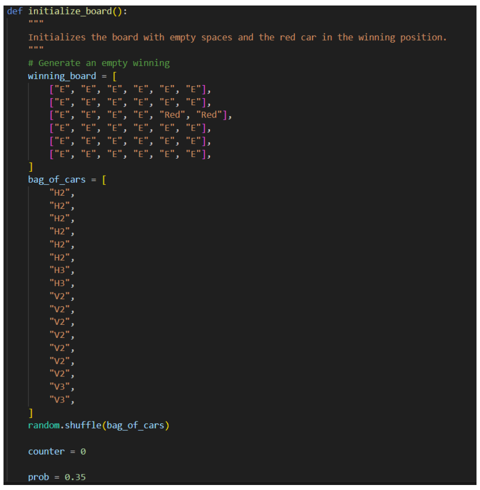
```python
def new_board(prev_board, direction, move_x, move_y):
    """
    Creates a new board after a move has been made.
    """

    filled_x = move_x
    filled_y = move_y

    next_board = prev_board
    if direction == "Right":
        if prev_board[move_y][move_x] == "H2":
            next_board[move_y][move_x - 1] = "E"
            next_board[move_y][move_x + 1] = "H2"

        elif prev_board[move_y][move_x] == "Red":
            next_board[move_y][move_x - 1] = "E"
            next_board[move_y][move_x + 1] = "Red"
        elif prev_board[move_y][move_x] == "H3":
            next_board[move_y][move_x - 2] = "E"
            next_board[move_y][move_x + 1] = "H3"

        filled_x = move_x + 1

    elif direction == "Left":
        if prev_board[move_y][move_x] == "H2":
            next_board[move_y][move_x + 1] = "E"
            next_board[move_y][move_x - 1] = "H2"
        elif prev_board[move_y][move_x] == "Red":
            next_board[move_y][move_x + 1] = "E"
            next_board[move_y][move_x - 1] = "Red"
        elif prev_board[move_y][move_x] == "H3":
            next_board[move_y][move_x + 2] = "E"
            next_board[move_y][move_x - 1] = "H3"
```



```python
elif direction == "Up":
    if prev_board[move_y][move_x] == "V2":
        next_board[move_y + 1][move_x] = "E"
        next_board[move_y - 1][move_x] = "V2"
    elif prev_board[move_y][move_x] == "V3":
        next_board[move_y + 2][move_x] = "E"
        next_board[move_y - 1][move_x] = "V3"

    filled_y = move_y - 1

elif direction == "Down":
    if prev_board[move_y][move_x] == "V2":
        next_board[move_y - 1][move_x] = "E"
        next_board[move_y + 1][move_x] = "V2"
    elif prev_board[move_y][move_x] == "V3":
        next_board[move_y - 2][move_x] = "E"
        next_board[move_y + 1][move_x] = "V3"

    filled_y = move_y + 1

return next_board, filled_x, filled_y
```

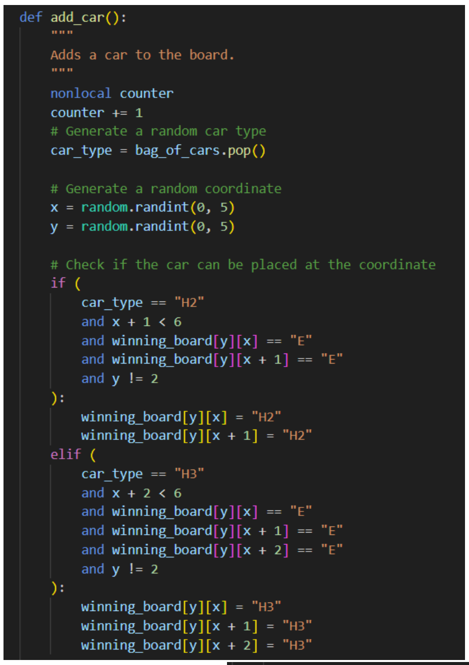
In the final stages of our project, we introduced an innovative board generation function, enhancing our model's capabilities. Previously, our approach relied on a hard-coded database of valid game boards sourced from the official Rush Hour website. Eager to advance our project, we developed a function that begins with a winning board state and then progressively adds vehicles, utilizing probabilistic methods, and employs our established propositions and constraints to simulate backward moves (which we called a *rewind*). This process continues until a valid initial game state is achieved, with the red car not in a winning position. While we recognize that this method may not produce the most intricate game boards, its incorporation marks a significant enhancement to our project. It not only demonstrates our model's versatility but also sets the stage for future improvements in board generation, possibly through more advanced algorithms or different computational techniques.

- First, we built a function that would initialize a random board, putting the red car in the winning position to ensure the game was solvable:

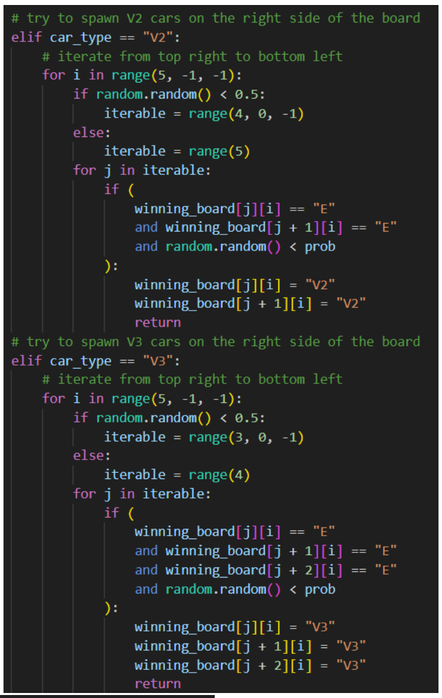


```python
def initialize_board():
    """
    Initializes the board with empty spaces and the red car in the winning position.
    """
    # Generate an empty winning
    winning_board = [
        ["E", "E", "E", "E", "E", "E"],
        ["E", "E", "E", "E", "E", "E"],
        ["E", "E", "E", "E", "Red", "Red"],
        ["E", "E", "E", "E", "E", "E"],
        ["E", "E", "E", "E", "E", "E"],
        ["E", "E", "E", "E", "E", "E"],
    ]
    bag_of_cars = [
        "H2",
        "H2",
        "H2",
        "H2",
        "H2",
        "H2",
        "H3",
        "H3",
        "V2",
        "V2",
        "V2",
        "V2",
        "V2",
        "V2",
        "V2",
        "V3",
        "V3",
    ]
    random.shuffle(bag_of_cars)

    counter = 0

    prob = 0.35
```

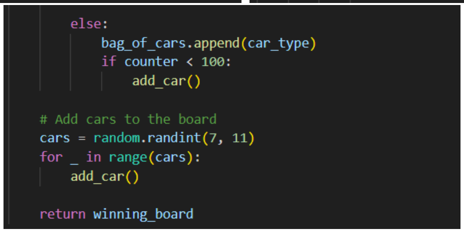
- From there, we were able to fill out the rest of the board with other cars:



```
def add_car():
    """
    Adds a car to the board.
    """
    nonlocal counter
    counter += 1
    # Generate a random car type
    car_type = bag_of_cars.pop()

    # Generate a random coordinate
    x = random.randint(0, 5)
    y = random.randint(0, 5)

    # Check if the car can be placed at the coordinate
    if (
        car_type == "H2"
        and x + 1 < 6
        and winning_board[y][x] == "E"
        and winning_board[y][x + 1] == "E"
        and y != 2
    ):
        winning_board[y][x] = "H2"
        winning_board[y][x + 1] = "H2"
    elif (
        car_type == "H3"
        and x + 2 < 6
        and winning_board[y][x] == "E"
        and winning_board[y][x + 1] == "E"
        and winning_board[y][x + 2] == "E"
        and y != 2
    ):
        winning_board[y][x] = "H3"
        winning_board[y][x + 1] = "H3"
        winning_board[y][x + 2] = "H3"
```

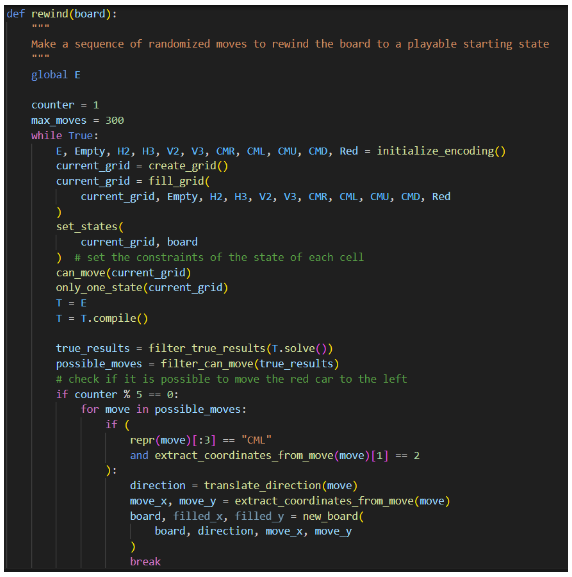


```
# try to spawn V2 cars on the right side of the board
elif car_type == "V2":
    # iterate from top right to bottom left
    for i in range(5, -1, -1):
        if random.random() < 0.5:
            iterable = range(4, 0, -1)
        else:
            iterable = range(5)
        for j in iterable:
            if (
                winning_board[j][i] == "E"
                and winning_board[j + 1][i] == "E"
                and random.random() < prob
            ):
                winning_board[j][i] = "V2"
                winning_board[j + 1][i] = "V2"
                return
# try to spawn V3 cars on the right side of the board
elif car_type == "V3":
    # iterate from top right to bottom left
    for i in range(5, -1, -1):
        if random.random() < 0.5:
            iterable = range(3, 0, -1)
        else:
            iterable = range(4)
        for j in iterable:
            if (
                winning_board[j][i] == "E"
                and winning_board[j + 1][i] == "E"
                and winning_board[j + 2][i] == "E"
                and random.random() < prob
            ):
                winning_board[j][i] = "V3"
                winning_board[j + 1][i] = "V3"
                winning_board[j + 2][i] = "V3"
                return
```

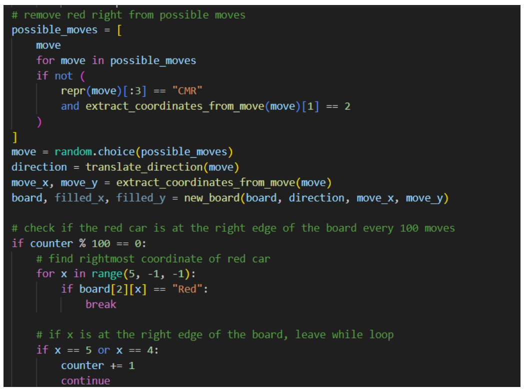


```
        else:
            bag_of_cars.append(car_type)
            if counter < 100:
                add_car()

    # Add cars to the board
    cars = random.randint(7, 11)
    for _ in range(cars):
        add_car()

    return winning_board
```

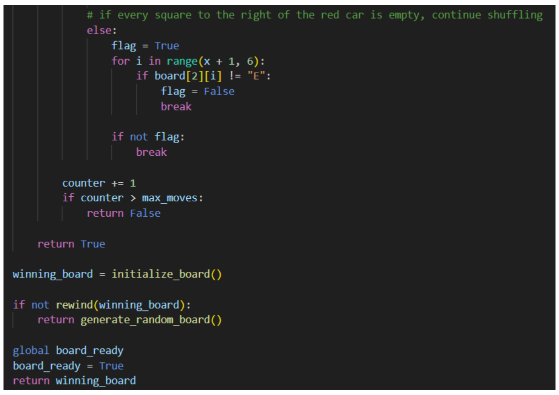
- Once our winning state was found, we could then '*rewind*' the board to a non-winning state so it would be playable:



```
def rewind(board):
    """
    Make a sequence of randomized moves to rewind the board to a playable starting state
    """
    global E

    counter = 1
    max_moves = 300
    while True:
        E, Empty, H2, H3, V2, V3, CMR, CML, CMU, CMD, Red = initialize_encoding()
        current_grid = create_grid()
        current_grid = fill_grid(
            current_grid, Empty, H2, H3, V2, V3, CMR, CML, CMU, CMD, Red
        )
        set_states(
            current_grid, board
        )  # set the constraints of the state of each cell
        can_move(current_grid)
        only_one_state(current_grid)
        T = E
        T = T.compile()

        true_results = filter_true_results(T.solve())
        possible_moves = filter_can_move(true_results)
        # check if it is possible to move the red car to the left
        if counter % 5 == 0:
            for move in possible_moves:
                if (
                    repr(move)[:3] == "CML"
                    and extract_coordinates_from_move(move)[1] == 2
                ):
                    direction = translate_direction(move)
                    move_x, move_y = extract_coordinates_from_move(move)
                    board, filled_x, filled_y = new_board(
                        board, direction, move_x, move_y
                    )
                    break
```



```
# remove red right from possible moves
possible_moves = [
    move
    for move in possible_moves
    if not (
        repr(move)[:3] == "CMR"
        and extract_coordinates_from_move(move)[1] == 2
    )
]
move = random.choice(possible_moves)
direction = translate_direction(move)
move_x, move_y = extract_coordinates_from_move(move)
board, filled_x, filled_y = new_board(board, direction, move_x, move_y)

# check if the red car is at the right edge of the board every 100 moves
if counter % 100 == 0:
    # find rightmost coordinate of red car
    for x in range(5, -1, -1):
        if board[2][x] == "Red":
            break

    # if x is at the right edge of the board, leave while loop
    if x == 5 or x == 4:
        counter += 1
        continue
```



```
            # if every square to the right of the red car is empty, continue shuffling
            else:
                flag = True
                for i in range(x + 1, 6):
                    if board[2][i] != "E":
                        flag = False
                        break

                if not flag:
                    break

        counter += 1
        if counter > max_moves:
            return False

    return True

winning_board = initialize_board()

if not rewind(winning_board):
    return generate_random_board()

global board_ready
board_ready = True
return winning_board
```

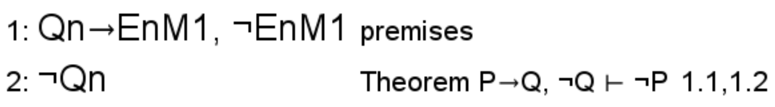
- All of this together resulted in a fascinating expansion to our original project that let us experiment with what it means for a board to be solvable in the game of Rush Hour.

Jape Proofs

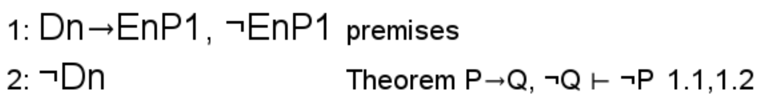
List the ideas you have to build sequents & proofs that relate to your project.

For the proofs, we will use E to represent *Cell Is Empty*, Q to represent *Cell Can Move Up*, D to represent *Cell Can Move Down*, R to represent *Cell Can Move Right*, and S to represent *Cell Can Move Left*. (Q and S were used as U and L were both unable to be identified in Jape).

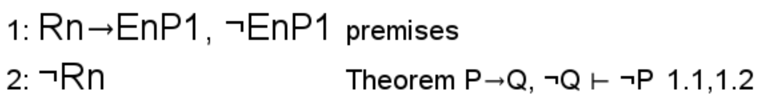
Proof Type 1: Can move up implies the spot above is empty; the spot above is not empty implies can not move up. The same is true for every direction.
nM1 represents n - 1 and nP1 represents n + 1



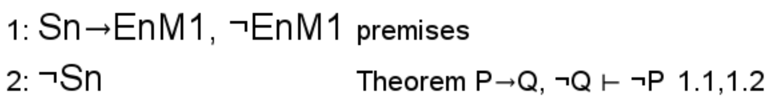
1: Qn→EnM1, ¬EnM1 premises
2: ¬Qn Theorem P→Q, ¬Q ⊢ ¬P 1.1,1.2



1: Dn→EnP1, ¬EnP1 premises
2: ¬Dn Theorem P→Q, ¬Q ⊢ ¬P 1.1,1.2



1: Rn→EnP1, ¬EnP1 premises
2: ¬Rn Theorem P→Q, ¬Q ⊢ ¬P 1.1,1.2

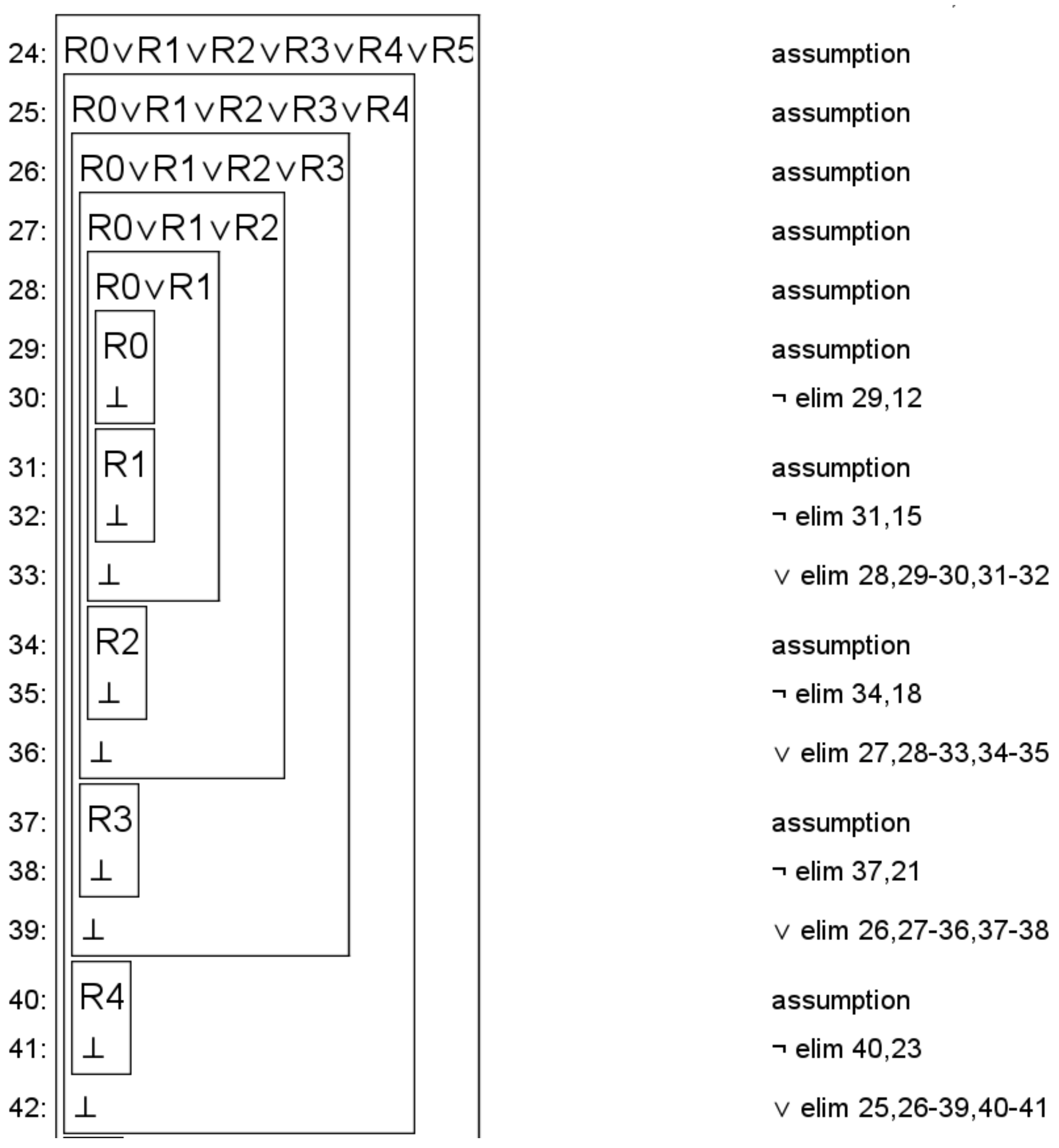


1: Sn→EnM1, ¬EnM1 premises
2: ¬Sn Theorem P→Q, ¬Q ⊢ ¬P 1.1,1.2

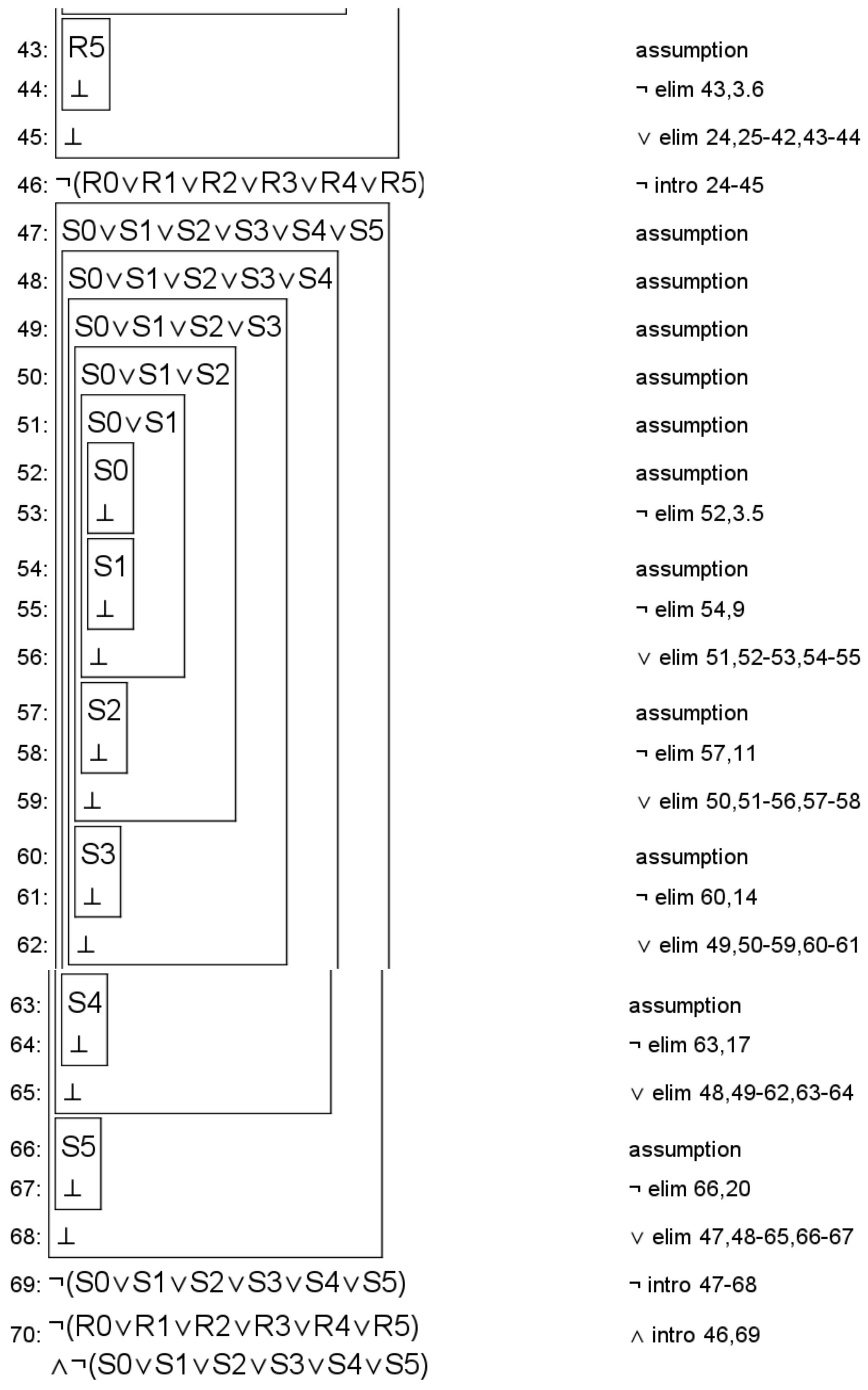
Proof 2: If every cell in a row is not empty, nothing can move left or right. For the case of simplicity, we will only show the number of the column as we are talking about cells in a single row.

Every cell is empty, Cell n can move right implies Cell n+1 is empty, Cell n can move left implies Cell n-1 is empty, Cell 0 can't move Left, Cell 5 can't move Right. IMPLIES No cells can move Right, and no cells can move Left.

1:	$\neg E0 \wedge \neg E1 \wedge \neg E2 \wedge \neg E3 \wedge \neg E4 \wedge \neg E5$, $R0 \to E1$	premises
2:	$R1 \to E2$, $R2 \to E3$, $R3 \to E4$, $R4 \to E5$, $S1 \to E0$	premises
3:	$S2 \to E1$, $S3 \to E2$, $S4 \to E3$, $S5 \to E4$, $\neg S0$, $\neg R5$	premises
4:	$\neg E0 \wedge \neg E1 \wedge \neg E2 \wedge \neg E3 \wedge \neg E4$	$\wedge$ elim 1.1
5:	$\neg E0 \wedge \neg E1 \wedge \neg E2 \wedge \neg E3$	$\wedge$ elim 4
6:	$\neg E0 \wedge \neg E1 \wedge \neg E2$	$\wedge$ elim 5
7:	$\neg E0 \wedge \neg E1$	$\wedge$ elim 6
8:	$\neg E0$	$\wedge$ elim 7
9:	$\neg S1$	Theorem $P \to Q, \neg Q \vdash \neg P$ 2.5,8
10:	$\neg E1$	$\wedge$ elim 7
11:	$\neg S2$	Theorem $P \to Q, \neg Q \vdash \neg P$ 3.1,10
12:	$\neg R0$	Theorem $P \to Q, \neg Q \vdash \neg P$ 1.2,10
13:	$\neg E2$	$\wedge$ elim 6
14:	$\neg S3$	Theorem $P \to Q, \neg Q \vdash \neg P$ 3.2,13
15:	$\neg R1$	Theorem $P \to Q, \neg Q \vdash \neg P$ 2.1,13
16:	$\neg E3$	$\wedge$ elim 5
17:	$\neg S4$	Theorem $P \to Q, \neg Q \vdash \neg P$ 3.3,16
18:	$\neg R2$	Theorem $P \to Q, \neg Q \vdash \neg P$ 2.2,16
19:	$\neg E4$	$\wedge$ elim 4
20:	$\neg S5$	Theorem $P \to Q, \neg Q \vdash \neg P$ 3.4,19
21:	$\neg R3$	Theorem $P \to Q, \neg Q \vdash \neg P$ 2.3,19
22:	$\neg E5$	$\wedge$ elim 1.1
23:	$\neg R4$	Theorem $P \to Q, \neg Q \vdash \neg P$ 2.4,22



```
24: R0∨R1∨R2∨R3∨R4∨R5                 assumption
25:   R0∨R1∨R2∨R3∨R4                  assumption
26:     R0∨R1∨R2∨R3                   assumption
27:       R0∨R1∨R2                    assumption
28:         R0∨R1                     assumption
29:           R0                      assumption
30:           ⊥                       ¬ elim 29,12
31:           R1                      assumption
32:           ⊥                       ¬ elim 31,15
33:         ⊥                         ∨ elim 28,29-30,31-32
34:         R2                        assumption
35:         ⊥                         ¬ elim 34,18
36:       ⊥                           ∨ elim 27,28-33,34-35
37:       R3                          assumption
38:       ⊥                           ¬ elim 37,21
39:     ⊥                             ∨ elim 26,27-36,37-38
40:     R4                            assumption
41:     ⊥                             ¬ elim 40,23
42:   ⊥                               ∨ elim 25,26-39,40-41
```

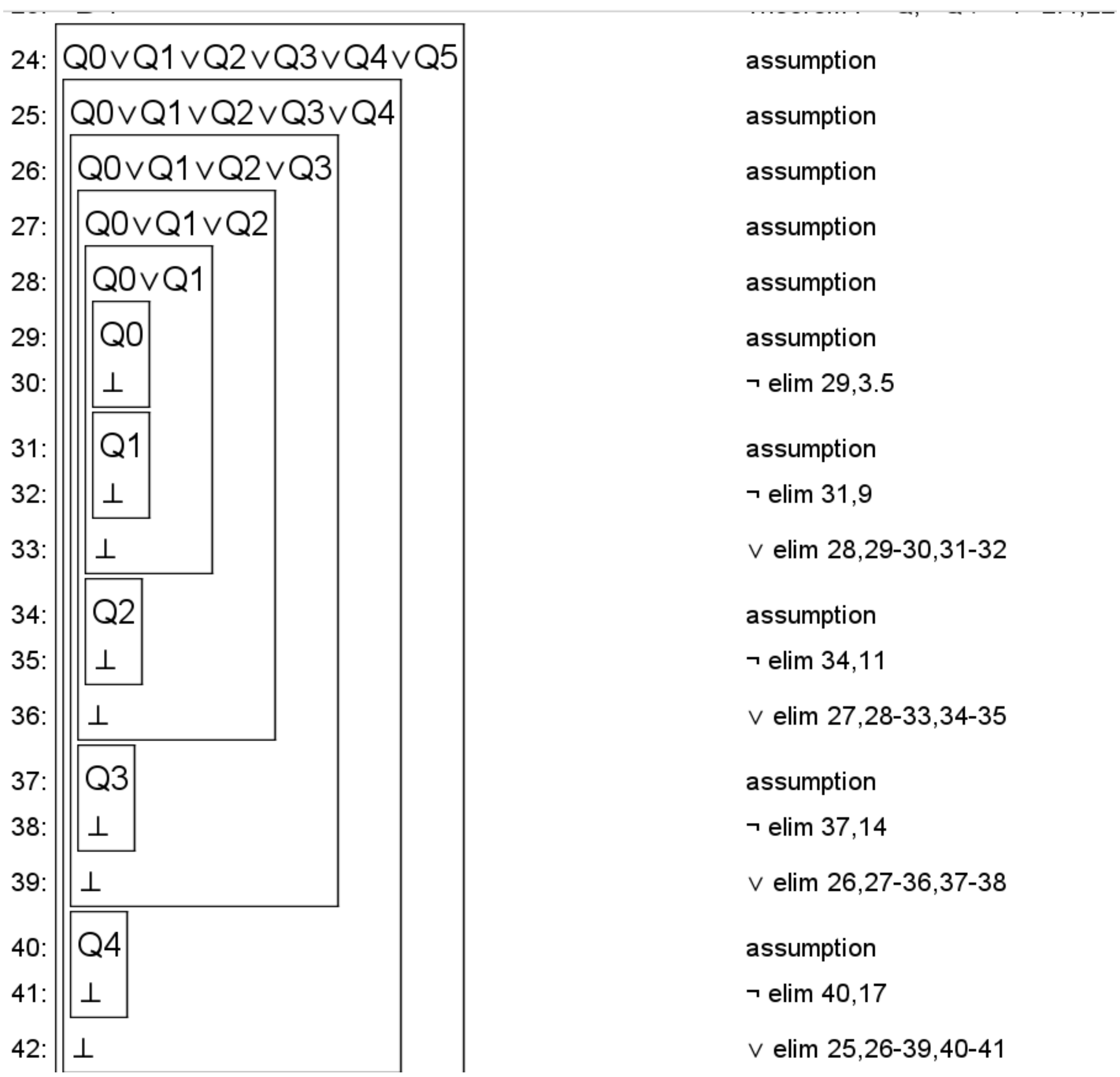


43:	$R5$	assumption
44:	$\perp$	¬ elim 43,3.6
45:	$\perp$	∨ elim 24,25-42,43-44
46:	$\neg(R0 \vee R1 \vee R2 \vee R3 \vee R4 \vee R5)$	¬ intro 24-45
47:	$S0 \vee S1 \vee S2 \vee S3 \vee S4 \vee S5$	assumption
48:	$S0 \vee S1 \vee S2 \vee S3 \vee S4$	assumption
49:	$S0 \vee S1 \vee S2 \vee S3$	assumption
50:	$S0 \vee S1 \vee S2$	assumption
51:	$S0 \vee S1$	assumption
52:	$S0$	assumption
53:	$\perp$	¬ elim 52,3.5
54:	$S1$	assumption
55:	$\perp$	¬ elim 54,9
56:	$\perp$	∨ elim 51,52-53,54-55
57:	$S2$	assumption
58:	$\perp$	¬ elim 57,11
59:	$\perp$	∨ elim 50,51-56,57-58
60:	$S3$	assumption
61:	$\perp$	¬ elim 60,14
62:	$\perp$	∨ elim 49,50-59,60-61
63:	$S4$	assumption
64:	$\perp$	¬ elim 63,17
65:	$\perp$	∨ elim 48,49-62,63-64
66:	$S5$	assumption
67:	$\perp$	¬ elim 66,20
68:	$\perp$	∨ elim 47,48-65,66-67
69:	$\neg(S0 \vee S1 \vee S2 \vee S3 \vee S4 \vee S5)$	¬ intro 47-68
70:	$\neg(R0 \vee R1 \vee R2 \vee R3 \vee R4 \vee R5) \wedge \neg(S0 \vee S1 \vee S2 \vee S3 \vee S4 \vee S5)$	∧ intro 46,69

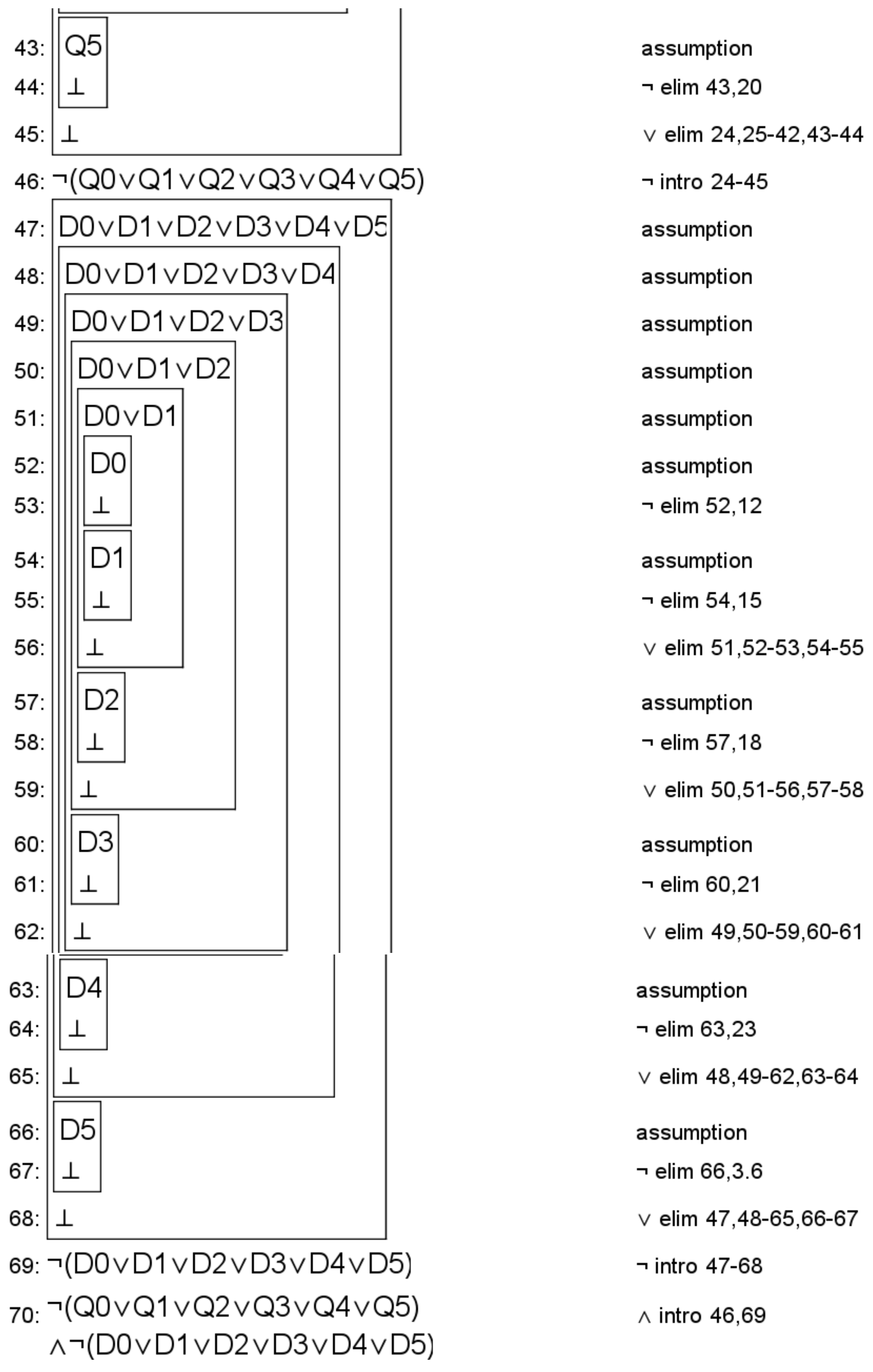
Proof 3: If every cell in a column is not empty, nothing can move up or down. For the case of simplicity, we will only show the number of rows as we are talking about cells in a single column.

Every cell is empty, Cell n can move down implies Cell n+1 is empty, Cell n can move up implies Cell n-1 is empty, Cell 0 can't move up, Cell 5 can't move down. IMPLIES No cells can move up, and no cells can move down.

1:	$\neg E0 \wedge \neg E1 \wedge \neg E2 \wedge \neg E3 \wedge \neg E4 \wedge \neg E5,\ D0 \rightarrow E1$	premises
2:	$D1 \rightarrow E2,\ D2 \rightarrow E3,\ D3 \rightarrow E4,\ D4 \rightarrow E5,\ Q1 \rightarrow E0$	premises
3:	$Q2 \rightarrow E1,\ Q3 \rightarrow E2,\ Q4 \rightarrow E3,\ Q5 \rightarrow E4,\ \neg Q0,\ \neg D5$	premises
4:	$\neg E0 \wedge \neg E1 \wedge \neg E2 \wedge \neg E3 \wedge \neg E4$	$\wedge$ elim 1.1
5:	$\neg E0 \wedge \neg E1 \wedge \neg E2 \wedge \neg E3$	$\wedge$ elim 4
6:	$\neg E0 \wedge \neg E1 \wedge \neg E2$	$\wedge$ elim 5
7:	$\neg E0 \wedge \neg E1$	$\wedge$ elim 6
8:	$\neg E0$	$\wedge$ elim 7
9:	$\neg Q1$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 2.5,8
10:	$\neg E1$	$\wedge$ elim 7
11:	$\neg Q2$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 3.1,10
12:	$\neg D0$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 1.2,10
13:	$\neg E2$	$\wedge$ elim 6
14:	$\neg Q3$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 3.2,13
15:	$\neg D1$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 2.1,13
16:	$\neg E3$	$\wedge$ elim 5
17:	$\neg Q4$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 3.3,16
18:	$\neg D2$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 2.2,16
19:	$\neg E4$	$\wedge$ elim 4
20:	$\neg Q5$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 3.4,19
21:	$\neg D3$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 2.3,19
22:	$\neg E5$	$\wedge$ elim 1.1
23:	$\neg D4$	Theorem $P \rightarrow Q,\ \neg Q \vdash \neg P$ 2.4,22



```
24: | Q0∨Q1∨Q2∨Q3∨Q4∨Q5          assumption
25: | | Q0∨Q1∨Q2∨Q3∨Q4          assumption
26: | | | Q0∨Q1∨Q2∨Q3          assumption
27: | | | | Q0∨Q1∨Q2          assumption
28: | | | | | Q0∨Q1          assumption
29: | | | | | | Q0          assumption
30: | | | | | | ⊥           ¬ elim 29,3.5
31: | | | | | | Q1          assumption
32: | | | | | | ⊥           ¬ elim 31,9
33: | | | | | ⊥             ∨ elim 28,29-30,31-32
34: | | | | | Q2            assumption
35: | | | | | ⊥             ¬ elim 34,11
36: | | | | ⊥               ∨ elim 27,28-33,34-35
37: | | | | Q3              assumption
38: | | | | ⊥               ¬ elim 37,14
39: | | | ⊥                 ∨ elim 26,27-36,37-38
40: | | | Q4                assumption
41: | | | ⊥                 ¬ elim 40,17
42: | | ⊥                   ∨ elim 25,26-39,40-41
```



43:	$Q5$	assumption
44:	$\perp$	¬ elim 43,20
45:	$\perp$	∨ elim 24,25-42,43-44
46:	$\neg(Q0 \lor Q1 \lor Q2 \lor Q3 \lor Q4 \lor Q5)$	¬ intro 24-45
47:	$D0 \lor D1 \lor D2 \lor D3 \lor D4 \lor D5$	assumption
48:	$D0 \lor D1 \lor D2 \lor D3 \lor D4$	assumption
49:	$D0 \lor D1 \lor D2 \lor D3$	assumption
50:	$D0 \lor D1 \lor D2$	assumption
51:	$D0 \lor D1$	assumption
52:	$D0$	assumption
53:	$\perp$	¬ elim 52,12
54:	$D1$	assumption
55:	$\perp$	¬ elim 54,15
56:	$\perp$	∨ elim 51,52-53,54-55
57:	$D2$	assumption
58:	$\perp$	¬ elim 57,18
59:	$\perp$	∨ elim 50,51-56,57-58
60:	$D3$	assumption
61:	$\perp$	¬ elim 60,21
62:	$\perp$	∨ elim 49,50-59,60-61
63:	$D4$	assumption
64:	$\perp$	¬ elim 63,23
65:	$\perp$	∨ elim 48,49-62,63-64
66:	$D5$	assumption
67:	$\perp$	¬ elim 66,3.6
68:	$\perp$	∨ elim 47,48-65,66-67
69:	$\neg(D0 \lor D1 \lor D2 \lor D3 \lor D4 \lor D5)$	¬ intro 47-68
70:	$\neg(Q0 \lor Q1 \lor Q2 \lor Q3 \lor Q4 \lor Q5) \land \neg(D0 \lor D1 \lor D2 \lor D3 \lor D4 \lor D5)$	∧ intro 46,69

First-Order Extension

- Domain of Discourse
  - Objects for cell coordinates and state coordinates
  - Objects for cell states
  - The positive integers in a range of (0, 5)
- Predicates
  - Cell(i,j,s) - cell i,j has state s
  - Red(i,j) - cell i,j contains part of the red car
  - V2(i,j) - cell i,j contains part of a V2 car
  - V3(i,j) - cell i,j contains part of a V3 car
  - H2(i,j) - cell i,j contains part of an H2 car
  - H3(i,j) - cell i,j contains part of an H3 car
- Constraints
  - A cell must have exactly one value be true at any time:

    $$\forall i.\forall j.\forall x.\forall y.(Cell(i,j,x) \land Cell(i,j,y) \implies (x = y))$$

  - The red vehicle must be on the third row of the grid, aligned with the exit:

    $$\forall j.\exists i.(Red(i,j) \implies (i = 2))$$

  - The game board must have dimensions of 6x6:

    $$\forall i.\forall j.(0 \leq i, j \leq 5)$$

  - Vehicles must stay within the boundaries of the game board and cannot leave it:

    $$\forall s.(Cell(i,j,s) \implies (0 \leq i, j \leq 5))$$

  - No vehicles moving in the same direction as the red car can be in the same row as the red car:

    $$\forall j.\neg\exists i.(R(i,j) \land H2(i,j) \land H3(i,j))$$

  - When one part of a car moves, the other parts of the car must move the same amount in the same direction:
    * For V2 vehicles:

      $$\forall i.\forall j.((V2(i,j) \land V2(i,j+1)) \implies (V2(i,j+1) \land V2(i,j+2)))$$

    * For V3 vehicles:

      $$\forall i.\forall j.((V3(i,j) \land V3(i,j+1) \land V3(i,j+2)) \implies (V3(i,j+1) \land V3(i,j+2) \land V3(i,j+3)))$$

    * For H2 vehicles (also applies to the Red car):

      $$\forall i.\forall j.((H2(i,j) \land H2(i,j+1)) \implies (H2(i,j+1) \land H2(i,j+2)))$$

    * For H3 vehicles:

      $$\forall i.\forall j.((H3(i,j) \land H3(i,j+1) \land H3(i,j+2)) \implies (H3(i,j+1) \land H3(i,j+2) \land H3(i,j+3)))$$

  - The game is won when the red car reaches the exit at cell (2,5). This condition can be represented in first-order logic as:

    $$Red(2,5) \rightarrow \text{Game Won}$$

- Theorems
  - If $R(2,5)$ is true, then the game is complete.
  - If there is any car that is not the red car occupying the third row (i=2), then the game is not currently winnable.
  - If every cell in a row is occupied by a vehicle, then no horizontal moves can be made in that row.
  - If every cell in a column is occupied by a vehicle, then no vertical moves can be made in that column.

Conclusion

In conclusion, our project successfully models the game 'Rush Hour' using logical propositions and constraints. For future work, we could explore using logic to generate more challenging game boards. Additionally, there's potential to develop a 'Rush Hour Solver' employing techniques from Reinforcement Learning (RL) and Machine Learning (ML). This approach could automate solving complex game scenarios, demonstrating the practical application of these advanced computational methods in problem-solving.