PIA: Week 2 Deliverable

Two Woman four men

May 2025

Model Redesign

During the development process, we decided to completely replace the original model with a new, more robust formulation, which has already been formally. From this new version, we proceeded with the model's coding and implementation.

Mathematical Model

We propose a Mixed Integer Linear Programming (MILP) model to determine how to optimally play tiles from the hand in a Rummikub board configuration, maximizing the number of tiles placed on the board.

Sets and Indices

- I: set of colors (e.g., $I = \{Red, Blue, Yellow, Black\}$)
- J: set of tile values (typically $J = \{1, 2, \ldots, 13\}$)
- K: the set of all possible valid combinations (either groups or runs) that can be played according to Rummikub rules.
- $C_k \subseteq I \times J$: for each $k \in K$, C_k is the set of tile positions (as ordered pairs (i, j)) involved in combination k.

  Each pair (i, j) represents a tile with color i and number j that would be used to form combination k. For example:

  - Group example: $C_k = \{(1, 5), (2, 5), (3, 5)\}$ represents a group of number 5 in three different colors.
  - Run example: $C_k = \{(1, 4), (1, 5), (1, 6)\}$ represents a run in color 1 (e.g., red) with numbers 4, 5, 6.

Parameters

- $h_{ij} \in \{0, 1, 2\}$: number of real tiles of type (i, j) in the player's hand
- $c \in Z_{\geq 0}$: number of jokers in the player's hand
- b_{ij}: number of real tiles already placed on the board at position (i, j)
- j_{ij}: number of jokers currently occupying position (i, j) on the board
- $bij_{ij} = (b_{ij}, j_{ij})$: tuple representing the tile content of the board at each position

Decision Variables

- $x_k \in \{0, 1\}$: 1 if combination k is played, 0 otherwise
- $u_{ijk} \in \{0, 1\}$: 1 if tile (i, j) from the hand is used in combination k
- $v_k \in Z_{\geq 0}$: number of jokers assigned to combination k

Objective Function

$$\max \sum_{k \in K} \left(\sum_{(i,j) \in C_k} u_{ijk} + v_k \right) \tag{1}$$

The goal is to maximize the total number of tiles (real and jokers) placed on the board.

Constraints

(C1) Tile Usage Constraint:

$$\sum_{k \in K} u_{ijk} \leq h_{ij} \quad \forall i \in I, j \in J \tag{2}$$

Each tile from the hand can be used at most once.

(C2) Joker Availability Constraint:

$$\sum_{k \in K} v_k \leq c \tag{3}$$

The total number of jokers used cannot exceed those available in the player's hand.

(C3) Combination Validity Constraint:

$$\sum_{(i,j) \in C_k} u_{ijk} + v_k \geq 3 \cdot x_k \quad \forall k \in K \tag{4}$$

A combination is only valid if it uses at least 3 tiles in total (real or jokers), which is the minimum required for valid groups or runs in Rummikub.

1 Mathematical Model Implementation

The optimization model for the Rummikub game was implemented using Python and the `PuLP` library for mixed-integer programming. The objective is to maximize the number of tiles placed on the board by forming valid groups or runs according to official game rules.

Data Structures

- `h[i][j]`: a matrix representing the player's hand. Each row corresponds to a color, and each column to a tile number (1 to 13).
- `bij[(i, j)]`: a dictionary that represents the current state of the board. The keys are tuples (color, number), and the values are pairs `(real tiles, jokers)`.
- `K_list`: a list of all possible valid combinations (groups and runs) generated from the rules.
- `x[k]`: binary decision variable indicating if combination `k` from `K_list` is used.
- `u[i][j]`: integer variable indicating how many real tiles of color `i` and number `j` are used from the hand.
- `v[i][j]`: integer variable for the number of jokers used as tile (i, j).

2 Input Validation and Parameter Checking

Before solving the model, the code performs rigorous validation of all input structures. This ensures the internal consistency of the data and avoids runtime errors due to malformed inputs. The validation is divided into three parts: the hand matrix, the board dictionary, and the number of jokers.

- **Hand matrix** `h`:
  - Must be a 4x13 matrix (4 colors and 13 numbers).
  - Each cell must contain an integer between 0 and 2 (max two tiles per cell).
  - No combination validation is done here—unmatched tiles are allowed.
- **Board dictionary** `bij`:
  - Each key is a tuple `(color, number)`.
  - Each value is a tuple `(t, jk)` where `t` is the number of real tiles, and `jk` is the number of jokers.
  - No cell may contain more than two total tiles.

  - Valid combinations must be either groups or sequences:
    * **Groups** must have tiles of the same number but different colors.
    * **Sequences** must be of the same color and consecutive numbers (or filled with jokers).

- **Number of jokers** `c`:
  - Must be an integer between 0 and 2.

These checks are implemented in the functions `validar_mano`, `validar_tablero`, and `validar_datos_completos`, which raise descriptive error messages if any condition is violated. Upon successful validation, a confirmation message is printed for each component.

Implementation in Colab

2.1 Validation Examples

To illustrate the validation process, we provide several test cases, including valid and invalid configurations. These help verify that the model correctly accepts or rejects inputs based on the defined rules.

2.1.1 Valid Example 1: Sequence and Group

This example includes on board:

- A valid sequence: Green 2–3–4–5 (color 2, numbers 1–4).
- A valid group: Three 9s of different colors (Red, Blue, Black).

```
h = [[0]*13 for _ in range(4)]
h[2][1] = 1  # Green-2
h[2][2] = 1  # Green-3
h[2][3] = 1  # Green-4
h[2][4] = 1  # Green-5
h[0][8] = 1  # Red-9
h[1][8] = 1  # Blue-9
h[3][8] = 1  # Black-9

bij = {
    (2, 1): (1, 0), (2, 2): (1, 0), (2, 3): (1, 0), (2, 4): (0, 1),
    (0, 8): (1, 0), (1, 8): (1, 0), (3, 8): (1, 0)
}

c = 0
validar_datos_completos(h, bij, c)
```

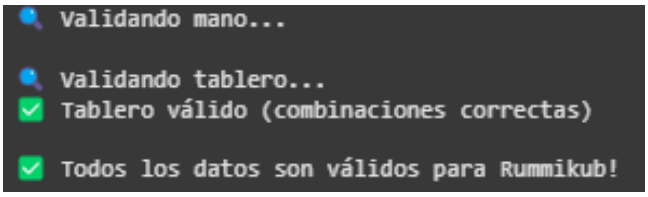

Figure 1: Output.

2.1.2 Valid Example 2: Sequence and Group

This example includes on board:

- A valid sequence: Green 2–3–4–5 (color 2, numbers 2–4–5).
- A valid group: Red 9, Green 9, Black 9 (colors 0, 2, 3, number 9).

```
h_ejemplo = [[0]*13 for _ in range(4)]

# Secuencia Verde 2{3{4
h_ejemplo[2][1] = 1  # Verde-2
h_ejemplo[2][2] = 1  # Verde-3
h_ejemplo[2][3] = 1  # Verde-4

# Grupo de 9s
h_ejemplo[0][8] = 1  # Rojo-9
h_ejemplo[1][8] = 1  # Azul-9
h_ejemplo[3][8] = 1  # Negro-9

# Tablero con agrupaciones
bij_ejemplo = {
    (2, 2): (1, 0),  # Verde-2
    (2, 4): (1, 0),  # Verde-4
    (2, 5): (1, 0),  # Verde-5

    (0, 9): (1, 0),  # Rojo-9
    (2, 3): (1, 1),  # Verde-3 (shared)
    (3, 9): (1, 0),  # Negro-9
}

c_ejemplo = 0  # Sin comodines

validar_datos_completos(h_ejemplo, bij_ejemplo, c_ejemplo)
```

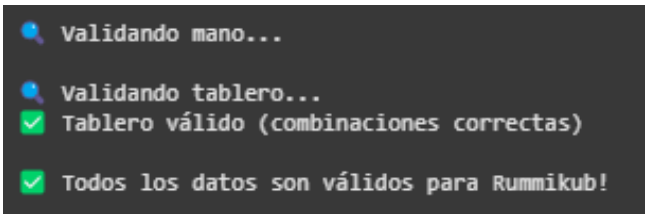

Figure 2: Output.

2.1.3 Invalid Example 1: Invalid Group, Invalid Sequence, and Invalid Cell

This example includes:

- A group with repeated colors (two red 5s).
- A sequence with a gap (not consecutive: Azul 2–4–5).
- A cell with more than 2 tiles (Verde-7 has 3 tiles).

```
h_invalido = [[0]*13 for _ in range(4)]

# Grupo inválido: dos fichas rojas con el mismo número (5)
h_invalido[0][4] = 1  # Rojo-5
h_invalido[0][4] += 1  # Otra vez Rojo-5 (misma ficha duplicada)

# Secuencia inválida: Azul 2{4{5 (falta el 3 y no hay comodines)
h_invalido[1][1] = 1  # Azul-2
h_invalido[1][3] = 1  # Azul-4
h_invalido[1][4] = 1  # Azul-5

# Celda inválida: Verde-7 tiene 3 fichas
h_invalido[2][6] = 3  # Verde-7 con 3 fichas

# Tablero con combinaciones inválidas
bij_invalido = {
    (0, 5): (1, 0),  # Grupo con colores repetidos
    (0, 5): (2, 0),  # Duplica celda

    # Secuencia no consecutiva
    (1, 2): (1, 0),
    (1, 4): (1, 0),
    (1, 5): (1, 0),

    # Celda con más de 2 fichas
    (2, 7): (3, 0),
```

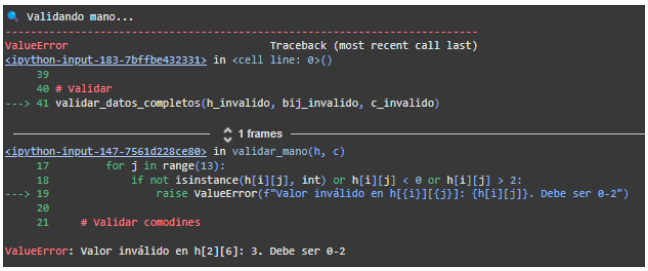

Figure 3: Output of invalid example.

```
}

c_invalido = 0  # Sin comodines

validar_datos_completos(h_invalido, bij_invalido, c_invalido)
```

2.1.4 Invalid Example 2: Non-consecutive Sequence

This example includes:

- A non-consecutive sequence: Verde 2–4–5 (missing 3, hence invalid).

```
h_ejemplo = [[0]*13 for _ in range(4)]

# Secuencia Verde 2{4{5 (no válida, falta el 3)
h_ejemplo[2][1] = 1  # Verde-2
h_ejemplo[2][3] = 1  # Verde-4
h_ejemplo[2][4] = 1  # Verde-5

# Tablero con agrupaciones
bij_ejemplo = {
    (2, 2): (1, 0),  # Verde-2
    (2, 4): (1, 0),  # Verde-4
    (2, 5): (1, 0),  # Verde-5
}

c_ejemplo = 0  # Sin comodines

validar_datos_completos(h_ejemplo, bij_ejemplo, c_ejemplo)
```

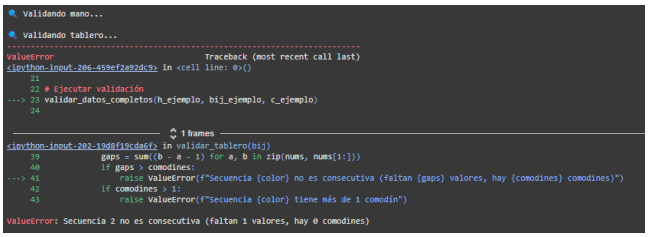

Figure 4: Output of invalid example.

3 Preliminary Model Testing

To verify the model's logic and feasibility, small test cases were created. These test cases help ensure that the model correctly handles different configurations and edge cases. Below are examples of several test cases, demonstrating how different hands and board configurations are validated.

3.1 Test Case 1: Basic Case

In this test case, a simple combination of tiles is tested. The hand contains the following tiles:

- Three Red tiles: 7, 8, and 9
- Three Blue tiles: 1, 2, and 3
- Three Green tiles: 4, 5, and 6

Additionally, the board configuration contains the following tiles already placed:

- Red 7, 8, 9
- Blue 1, 2, 3
- Green 4, 5, 6

No wildcard is used in this test case.

```
def test_basic_case1():
    # Define the hand of tiles
    h_test = [[0]*13 for _ in range(4)]  # 4 colors, 13 numbers

    # Place Red 7, 8, 9
    h_test[0][6] = 1  # Red 7
    h_test[0][7] = 1  # Red 8
    h_test[0][8] = 1  # Red 9
```

```
# Place Blue 1, 2, 3
h_test[1][0] = 1  # Blue 1
h_test[1][1] = 1  # Blue 2
h_test[1][2] = 1  # Blue 3

# Place Green 4, 5, 6
h_test[2][3] = 1  # Green 4
h_test[2][4] = 1  # Green 5
h_test[2][5] = 1  # Green 6

c_test = 0  # No wildcards

# Create the board configuration (bij_test)
bij_test = {
    (0, 7): (1, 0),  # Red 7
    (0, 8): (1, 0),  # Red 8
    (0, 9): (1, 0),  # Red 9
    (1, 0): (0, 1),  # Blue 1
    (1, 1): (0, 1),  # Blue 2
    (1, 2): (0, 1),  # Blue 3
    (2, 3): (0, 1),  # Green 4
    (2, 4): (0, 1),  # Green 5
    (2, 5): (0, 1)   # Green 6
}

# Call the model creation function and solve
model, K, x, u, v = create_model(h_test, bij_test, c_test)
model.solve()
```

Figure 5: Output.

3.2 Test Case 2: Mixed Valid Combinations (No Wildcard)

In this test case, the hand contains a mix of potential sets and runs using the four colors: Red, Blue, Black, and Orange. The hand contains the following tiles:

- Red tiles:7, 8, 9, 10
- Blue tiles: 1, 7, 8, 12
- Black tiles: 1, 5, 9
- Orange tiles: 1, 7, 9

Additionally, the board configuration contains the following tiles already placed:

- Red 10, Blue 10, Black 10 (Set)
- Orange 6, 7, 8 (Run)

No wildcard is used in this test case.

```
def test_caso_basico2():
    h_test = [[0]*13 for _ in range(4)]  # 4 colores, 13 números

    h_test[1][0] = 1  # Blue 1
    h_test[2][0] = 1  # Black 1
    h_test[3][0] = 1  # Orange 1
    h_test[3][6] = 1  # Orange 7
    h_test[0][8] = 1  # Red 9
    h_test[0][9] = 1  # Red 10
    h_test[2][8] = 1  # Black 9
    h_test[3][8] = 1  # Orange 9
    h_test[2][4] = 1  # Black 5
    h_test[1][6] = 1  # Blue 7
    h_test[1][7] = 1  # Blue 8
    h_test[0][6] = 1  # Red 7
    h_test[0][7] = 1  # Red 8
    h_test[0][8] = 1  # Red 9
    h_test[1][11] = 1 # Blue 12

    c_test = 0  # No wildcard

    bij_test = {
        (0, 9): (1, 0),  # Red 10
        (1, 9): (0, 1),  # Blue 10
        (2, 9): (1, 0),  # Black 10
        (3, 5): (1, 0),  # Orange 6
```

```
    (3, 6): (1, 0),  # Orange 7
    (3, 7): (1, 0)   # Orange 8
}

model, K, x, u, v = create_model(h_test, bij_test, c_test)
model.solve()
```

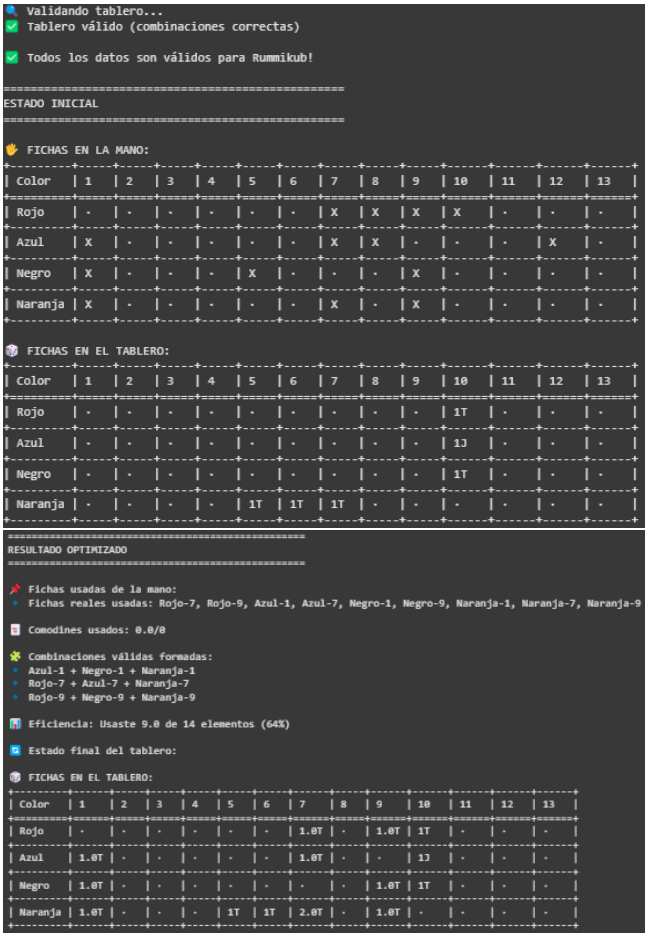

Figure 6: Output for Test Case 2.

3.3 Test Case 3: Similar to Test Case 2 with Fewer Tiles

This test case is a variation of Test Case 2, where a few tiles were removed from the hand to test the model's sensitivity and optimization under a slightly different configuration. The hand contains:

- Red tiles: 7, 8, 10
- Blue tiles: 1, 7, 8, 12
- Black tiles: 1, 5, 9
- Orange tiles: 1, 7, 9

The board remains the same as in Test Case 2.

```
def test_caso_basico3():
    h_test = [[0]*13 for _ in range(4)]

    h_test[1][0] = 1  # Blue 1
    h_test[2][0] = 1  # Black 1
    h_test[3][0] = 1  # Orange 1
    h_test[3][6] = 1  # Orange 7
    h_test[0][9] = 1  # Red 10
    h_test[2][8] = 1  # Black 9
    h_test[3][8] = 1  # Orange 9
    h_test[2][4] = 1  # Black 5
    h_test[1][6] = 1  # Blue 7
    h_test[1][7] = 1  # Blue 8
    h_test[0][6] = 1  # Red 7
    h_test[0][7] = 1  # Red 8
    h_test[1][11] = 1 # Blue 12

    c_test = 0  # No wildcard

    bij_test = {
        (0, 9): (1, 0),  # Red 10
        (1, 9): (0, 1),  # Blue 10
        (2, 9): (1, 0),  # Black 10
        (3, 5): (1, 0),  # Orange 6
        (3, 6): (1, 0),  # Orange 7
        (3, 7): (1, 0)   # Orange 8
    }

    model, K, x, u, v = create_model(h_test, bij_test, c_test)
    model.solve()
```

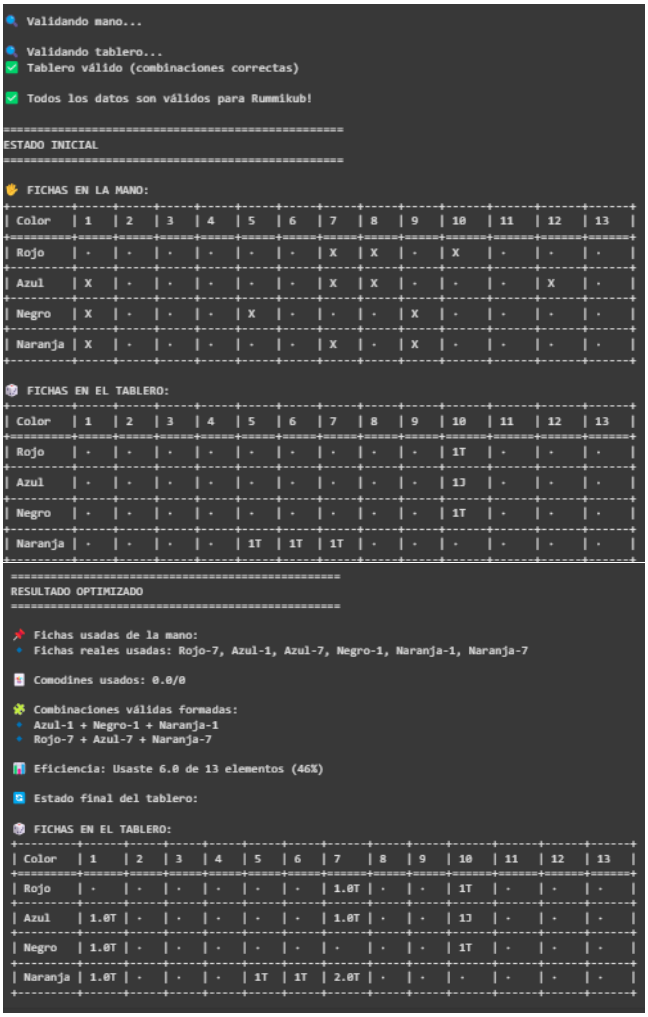

Figure 7: Output for Test Case 3.

4 Conclusion

The developed model successfully validates and solves basic Rummikub scenarios. It has been verified that, in simple cases, the model correctly identifies valid combinations, whether they are groups (same number, different colors) or runs (consecutive numbers of the same color), and it utilizes the highest possible number of tiles from the hand.

However, some limitations were identified when dealing with more complex combinations, particularly runs involving four or more consecutive elements. In these situations, the model often fails to correctly form the complete sequence, indicating an area for improvement in handling extended runs.

Validation examples have been added to demonstrate these behaviors. The system is ready to receive further instructions for adjustments, enhancements, or additional testing to improve its robustness and applicability in more realistic scenarios.