Universiteit Leiden
The Netherlands

Opleiding Informatica

Algorithms for

Rummikub Puzzles

Miguel van der Wekken

Supervisors:
dr. W.A. Kosters & dr. J.K. Vis

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

June 26, 2025

Abstract

A Rummikub puzzle is a one-player variant of the Rummikub game. The objective is to form valid groups and runs to maximize the total points, which are determined by the value of the tiles used in these valid groups and runs. This thesis addresses the optimization problem of solving Rummikub puzzles, with a focus on reducing the memory usage and categorizing puzzle difficulty. To achieve this, a state-encoding mechanism is introduced to prevent redundant computations, which is extended by dynamic memory allocation to only handle reachable states. Additionally, a set of pre-computable features is proposed to categorize the difficulty of Rummikub puzzles.

Contents

1 Introduction

The game of RUMMIKUB is a tile-based game for two to four players that combines elements of strategy and luck. To provide a clear understanding of its rules, we first outline how the game is played based on the official rules [Lem].

1.1 Rules and Game Play

RUMMIKUB uses 106 tiles (or playing cards); 104 of them are numbered from 1 to 13, and are colored by one of the four colors in the game. For this thesis, we use {black, green, red, yellow} as available colors. There is exactly one duplicate of every number-color combination in the game. The remaining two tiles are jokers, which can substitute for any other tile.

The game starts with every player drawing 14 tiles. The remaining tiles are referred to as the *pool*. The players then take turns in which they can either draw a tile from the pool and add it to their current hand or play a subset of the tiles in their hand (if they are allowed to do so according to the rules described later). During the game, the players are not able to see the hands of other players. This makes RUMMIKUB an imperfect information game, similar to Poker, as players do not have access to all information, such as the hands of opponents.
The player can play their tiles in two ways: runs or groups. A *run* (also called street) consists of three or more consecutive numbers of the same color, and a *group* is defined as 3 or 4 tiles with the same number but consisting of different colors. For the initial meld, which is the first valid sets of tiles a player lays down, the numbers on the tiles of the player must add up to 30 or more points. The player can achieve this by forming any number of valid runs and/or groups, as long as they total at least 30 points and come from their hand. After this initial play, the player is free to manipulate the tiles on the table with their own tiles as long as the table at the end of the player's turn still consists of valid runs and groups and the tiles that were on the table before their turn must still be present on the table afterward. If a player fails to play one of their hand's tiles on the table in their turn, they have to take a tile from the pool. The objective of the game is to be the first player to play all their tiles as valid runs and/or groups on the table. If the pool becomes empty during the game, and no player can make a valid move, the player with the fewest points remaining in their hand wins. When playing multiple rounds of RUMMIKUB, players count the points remaining in their hands at the end of each game, which are then added to their total score. The player with the lowest score after all rounds wins.

In Figure 1, an example of a possible valid run is shown. This run consists of tiles numbered from 4–9 of the same color (black/clubs). A player could manipulate this run by taking tiles at each end of the run or splitting the run into multiple runs, as long as the created run(s) are still valid. In Figure 2, a possible manipulation of the previous run is shown. For this manipulation, a player used the black/clubs four tile to make a group of fours and the black/clubs nine tile to make a new run with the duplicated eight tile. The previous run now becomes a run of 5–8.

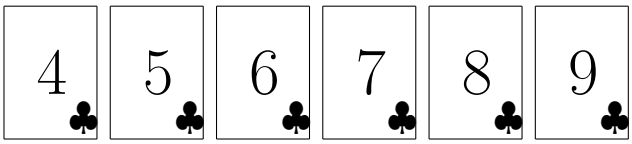

Figure 1: An example run laid on the table at the beginning of a player's turn.

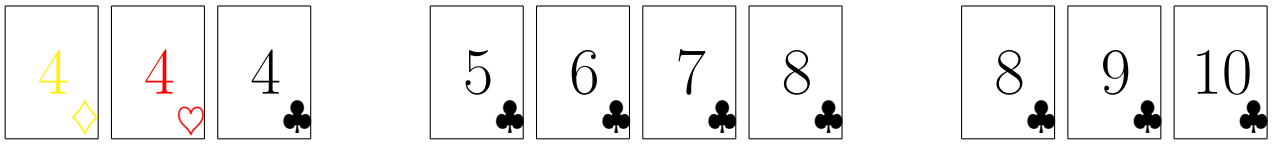

Figure 2: A possible manipulation of the run using a subset of the player's tiles, where the hearts, diamonds and clubs indicate different color tiles.

In this thesis, we study a one-player variant of the RUMMIKUB game, which we will refer to as the RUMMIKUB *Puzzle*. Unlike the standard version, this variant has no turns and no pool of tiles to draw from. Instead, the player has a fixed set of tiles to form valid groups and runs according to the game rules. The goal of the puzzle is to maximize the total points, which is defined as the sum of the values of all tiles used in valid groups and runs. The puzzle becomes an optimization problem for a given tile configuration.

1.2 Research Question

Given our RUMMIKUB puzzle definition, this thesis addresses the following research questions:

1. How can we reduce the memory usage by encoding RUMMIKUB puzzle states, based on the initial puzzle configuration?

2. How can we attach a difficulty measure to a given RUMMIKUB puzzle?

1.3 Thesis Overview

The remainder of this thesis is organized as follows. Section 2 formally defines the RUMMIKUB puzzle and introduces the notation used throughout this thesis. Section 3 reviews research that has been conducted on the game of RUMMIKUB, including early work using Integer Linear Programming, computer vision applications, and one-player puzzle variants. Section 4 presents a naive brute-force approach to solving the puzzle, followed by some optimizations. Section 5 discusses memory usage and introduces a memory-efficient representation. Section 6 introduces upper and lower bounds for RUMMIKUB puzzles, as well as techniques for reducing the problem size. Section 7 introduces a classifier that predicts the difficulty of a RUMMIKUB puzzle based on features from the initial configuration.
This research is part of the bachelor project of the Computer Science program at LIACS, Leiden University, under the supervision of dr. W.A. Kosters and dr. J.K. Vis.

2 Definitions

In this section, we formally define the RUMMIKUB *puzzle* using the following parameters:

- Let $k > 0$ be the number of distinct colors (or suits, if using cards instead of tiles).
- Let $n > 0$ be the maximum value a tile can take.
- Let $m > 0$ be the maximum number of copies available for each tile. Thus, for every color-value pair (c, v), at most m copies are present.
- Let $s \geq 3$ be the minimal set size required to form valid groups and runs.
- Let $j \geq 0$ be the number of joker tiles, which can substitute for any color-value pair (c, v).

In order to avoid additional complexity, we will fix $j = 0$ (no joker tiles) and the minimal set size for both groups and runs as $s = 3$ in this thesis.
Given these parameters, we define the universal tile set as:

$$T(k, m, n) = \bigcup_{c \in C, v \in V} \{(c, v)_i \mid 1 \leq i \leq m\}.$$

Here:

- $C = \{1, 2, \ldots, k\}$ is the set of colors.
- $V = \{1, 2, \ldots, n\}$ is the set of possible values.

A RUMMIKUB puzzle instance $G(k, m, n)$ is then defined as a subset of $T(k, m, n)$:

$$G(k, m, n) \subseteq T(k, m, n).$$

The RUMMIKUB puzzle can now be stated as an optimization problem:

> Given a configuration $G(k, m, n)$ of the RUMMIKUB puzzle, with parameters k, m and n, what is the maximum number of points that can be obtained by forming valid groups and runs?

The points are calculated as the sum of all the tile values included in the valid groups and runs.

Another approach to the RUMMIKUB puzzle could be to minimize the number of *unused* tiles. These tiles are not included in the valid groups or runs formed, given a RUMMIKUB puzzle configuration G. However, for this thesis, maximizing the number of points is chosen since, for the original RUMMIKUB game, the points on the remaining tiles matter. That being said, the algorithms in this thesis can easily be adapted to minimize the number of unused tiles. This can be achieved by assigning a value of one to each tile instead of the actual tile values.

3 Related Work

Compared to other combinatorial games such as Chess or Poker, there is limited research on the game of Rummikub.
The earliest research on Rummikub is by Den Hertog and Hulshof [dHH06]. Their research question aligns closely with ours: What is the maximum number or value of the tiles you can place on the table? To address this question, they formulated the problem as an Integer Linear Programming (ILP) model. ILP is a mathematical optimization technique that finds the best solution under a set of linear constraints, with the decision variables being integers. The ILP model was later extended with constraints to minimize the change to the existing sets on the board. This produced optimal solutions with minimal changes to runs and groups on the table.

The most recent research [VMLV25] about Rummikub has been conducted in the field of Computer Vision. In this study, the authors evaluate a method in which an Artificial Neural Network is combined with background knowledge and reasoning through the IDP-Z3 system in order to improve the detection and validation of Rummikub game states. The authors showed that for this Rummikub problem, the background knowledge is worth as much as two-thirds of the data set or slightly more than half of the training time.

Further research has been done in the Reinforcement Learning field [KK03]. In this research, the authors compare the training of an agent with two methods: Temporal-Difference learning versus Evolutionary learning. The agents are learning to play Rummy, which is a 52-card game variant of the Rummikub game. After letting the different learning agents play against each other. The Evolutionary learning agent produced superior results.

Research more directly aligned with this thesis has been conducted on one-player Rummikub puzzles. In [vRTV15], the authors addressed the same Rummikub problem and used the same input parameters according to the above given definition of the Rummikub problem. Given a subset of the Rummikub tile set, form valid sets of runs and groups such that the score is maximized. The authors provided a polynomial algorithm with a dynamic programming approach to solve this Rummikub problem. In this paper, the problem of counting winning hands is also briefly researched.

A bachelor thesis [Fei23] further investigated the counting of winning Rummikub hands. The author proposed the CWH algorithm, which produces all winning tile configurations in the game of Rummikub. The problem, however, was excessive running time and memory usage. The author introduced some interesting optimizations, such as pruning and parallel processing, to tackle this problem.

Another thesis [Gul19] explored a recursive method and a heuristic guided state space search. However, due to restricted access, this thesis could not be examined in more detail.

4 Initial Approach

The first approach to solve the RUMMIKUB puzzle for a RUMMIKUB configuration: $G(k, m, n)$, is a brute-force method. Here, every possible combination of taken runs and groups is calculated to get the maximum number of points possible. First, all the tiles of the RUMMIKUB puzzle are stored in a two-dimensional array, where the first axis represents the tile number and the second axis is the tile color (suit). The value at any given position in this array corresponds to the number of copies of that given tile. The algorithm begins at the last row (row n) and iterates through all the possible combinations of runs for each column. The possible run sizes are $\{0, 3, 4, 5\}$. A run of size 0 represents an empty run, which is necessary to account for permutations where no runs are formed and only groups are used. Since the minimal set size is $s = 3$, only runs of length three or greater are considered. Runs of size six can be split into two runs of size three, so only runs sizes smaller than six are included. Generally, for arbitrary s we only consider run sizes in the range $\{0, s, \ldots, 2s - 1\}$, since any run of size $2s$ or larger can be split into two or more valid runs. After determining a combination of possible runs for the current row, the remaining tiles are used to calculate the group value for that row. Then, we recursively call this method on the row above until we reach the first row in which we know the total sum of all the tiles that are present in valid groups and runs according to the current configuration of runs and groups. The current configuration score is compared to the maximum score found. This will ensure that after executing the maximum number of points given a RUMMIKUB configuration is found.
This approach correctly answers some smaller test configurations of the RUMMIKUB puzzle $G(4, 2, 40)$. But for the larger test configurations $G(4, 2, 100)$, the algorithm failed since it is too computationally expensive. The problem with the first brute-force approach is it recursively calculates all possible configurations of runs and groups. As the number of tiles n or the number of copies m increases, the complexity grows exponentially.

4.1 Test Files

The test files created by the authors of [vRTV15] are used in this thesis to test the correctness of algorithms for the RUMMIKUB puzzle. In these test files, there are multiple RUMMIKUB puzzles defined and every puzzle consists of multiple RUMMIKUB tiles.

The test files are built as follows. The first line has the number of RUMMIKUB puzzles found in the test file. The following lines are the RUMMIKUB puzzles, which consist of two lines. The first indicates the number of tiles present in this puzzle. The second line has all these tiles, which are represented by the number of the tile followed by the first letter of the color of the tile (b = black, g = green, r = red and y = yellow), the tiles are separated by spaces.

An example of such a test file is the following:

```
2
28
1b 1b 1g 1y 1y 2b 2y 3b 3b 3g 3g 3r 3r 3y 3y 4b 4g 4g 4r 4r 5b 5b 5g 5g 5r 5r 5y 5y
23
1g 1r 1r 1y 2b 2g 2r 2r 2y 3b 3b 3g 3g 3r 3y 3y 4b 4g 4y 5b 5b 5g 5y
```

This gives the two RUMMIKUB puzzles from Figure 3. The input parameters for these RUMMIKUB puzzles are: $k = 4, m = 2, n = 5$.

row	b	g	r	y
1	2	1	0	2
2	1	0	0	1
3	2	2	2	2
4	1	2	2	0
5	2	2	2	2

(a) First RUMMIKUB puzzle.

row	b	g	r	y
1	0	1	2	1
2	1	1	2	1
3	2	2	1	2
4	1	1	0	1
5	2	1	0	1

(b) Second RUMMIKUB puzzle.

Figure 3: Two-dimensional representation of the two examples RUMMIKUB puzzles.

The output file in which the maximum number of points per RUMMIKUB puzzle is shown looks like this:

```
93
65
```

The largest test file has the following RUMMIKUB configuration: G(4, 2, 100), and has 10,000 RUMMIKUB puzzles.

4.2 Group Value

After forming runs from the current row, the algorithm calculates the current *group value*. The group value is the maximum number of points that can be obtained by forming valid groups given a set of tiles with the same number, which is a row in the two-dimensional representation of the RUMMIKUB puzzle. To compute this, the algorithm takes the number of occurrences of each tile in the row and sorts them in ascending order.
Table 1 and Table 2 list all possible combinations of tile counts in a row, representing how a specific number may occur across the different colors. For $m = 2$, there are 15 combinations and for $m = 3$, there are 35.
The total number of group value combinations can be computed. Given parameters k and m, we select k integers from the range 0 to m, allowing repetitions and disregarding the order. This corresponds to the number of combinations with repetition, given by the following formula:

$$\binom{m+k}{k}. \tag{1}$$

For instance, with $k = 4$ and $m = 2$, the number of combinations is:

$$\binom{2+4}{4} = \binom{6}{4} = \frac{6!}{4!(6-4)!} = 15,$$

which corresponds to the 15 combinations shown in Table 1.

Similarly, for $k = 4$ and $m = 3$, we get:

$$\binom{3+4}{4} = \binom{7}{4} = 35,$$

matching the 35 combinations in Table 2.

For calculating the group value with $m = 3$ (which also applies to $m = 2$) and $k = 4$, Algorithm 1 is used. The main idea of the algorithm is to compute the group value for a given combination without the need for a table storing all possible combinations and their corresponding group values. The counts of each tile color in a row are denoted as a, b, c and d, where $0 \leq a \leq b \leq c \leq d \leq m$. The key steps of the algorithm are as follows:

1. Case $a = 0$:
    - At least one color does not contain a tile for the current number/row.
    - The only valid group consists of the remaining three colors.
    - Since b is the smallest number of copies, the group value is $b \times 3$.
2. The total number of tiles n_{tiles} in the row is computed.
3. Check whether the group value equals the total number of tiles:
    - This holds if either $a \geq 2$ or $c = d$.
    - The only exception is the combination 1133. we ensure that $n_{\text{tiles}} \neq 8$.
4. The remaining cases:
    - Five specific combinations remain for which the group value is less than the number of tiles.
    - The algorithm identifies which of these applies and returns the corresponding group value.

4.3 Clearable Groups

A group/row is considered *clearable* if all the tiles in that group can be formed into valid groups. Equivalently, a group is clearable if its group value equals the total number of tiles in the row. To determine whether a given combination is clearable, we use a similar algorithm as for the group value, which is as follows:

1. Check if the total number of tiles is zero or greater than 8:
    - In this case, all the combinations are clearable, as can be seen in Table 2.
2. Otherwise check if $b = d$.
    - In this case, all the combinations are clearable except 1122.

3. The combination 1122 is also clearable, as we can form two valid groups of size three.

4. All other combinations are not clearable.

Algorithm 1 Algorithm to compute the group value for a combination with $k = 4$ and $m = 3$.

Require: Sorted vector (a, b, c, d), with $0 \leq a \leq b \leq c \leq d \leq 3$
  if $a = 0$ **then**
    return $b \times 3$
  end if
  $n_{\text{tiles}} \leftarrow a + b + c + d$
  if $a \geq 2$ **or** $(c = d$ **and** $n_{\text{tiles}} \neq 8)$ **then**
    return n_{tiles}
  end if
  if $c = 1$ **then** ▷ combinations: 1112 and 1113
    return 4
  end if
  if $b = 1$ **then** ▷ combinations: 1123 and 1133
    return 6
  end if
  return 7 ▷ combinations: 1223

Combinations	Number of tiles	Group value	Clearable?
0000	0	0	True
0001	1	0	False
0002	2	0	False
0011	2	0	False
0012	3	0	False
0022	4	0	False
0111	3	3	True
0112	4	3	False
0122	5	3	False
0222	6	6	True
1111	4	4	True
1112	5	4	False
1122	6	6	True
1222	7	7	True
2222	8	8	True

Table 1: Group values and clearability for all the non-decreasing combinations of tile counts with $k = 4$ and $m = 2$.

Permutation	Number of tiles	Group value	Clearable ?
0000	0	0	True
0001	1	0	False
0002	2	0	False
0003	3	0	False
0011	2	0	False
0012	3	0	False
0013	4	0	False
0022	4	0	False
0023	5	0	False
0033	6	0	False
0111	3	3	True
0112	4	3	False
0113	5	3	False
0122	5	3	False
0123	6	3	False
0133	7	3	False
0222	6	6	True
0223	7	6	False
0233	8	6	False
0333	9	9	True
1111	4	4	True
1112	5	4	False
1113	6	4	False
1122	6	6	True
1123	7	6	False
1133	8	6	False
1222	7	7	True
1223	8	7	False
1233	9	9	True
1333	10	10	True
2222	8	8	True
2223	9	9	True
2233	10	10	True
2333	11	11	True
3333	12	12	True

Table 2: Group value and clearability for all the different group permutations with $k = 4$ and $m = 3$.

4.4 Group Value for Variable Colors and Copies

Algorithm 1 applies to our RUMMIKUB configurations, specifically $G(4, 2, 100)$ and $G(4, 3, 100)$, where $k = 4$ and $m = 2$. However, this algorithm is designed for fixed values of k and m and cannot be generalized directly. To extend the group value computation to arbitrary values of k and m, we introduce a closed formula.
Let $c = (c_1, c_2, \ldots, c_k)$ be the sorted vector representing the number of tiles of each of the k colors for a given value. Then, c_i denotes the number of tiles of color i, such that the following holds:

$$0 \leq c_1 \leq c_2 \leq \ldots \leq c_{k-1} \leq c_k \leq m. \tag{2}$$

If we fix the minimal set size $s = 3$, we obtain the following closed formula:

$$\text{Group value} = \begin{cases} 0 & \text{if } k < 3, \\ 3 \times \sum_{i=1}^{k-2} c_i & \text{if } \sum_{i=1}^{k-2} c_i \leq c_{k-1}, \\ \sum_{i=1}^{k-1} c_i + \min\left(c_k, \left\lfloor \frac{\sum_{i=1}^{k-1} c_i}{2} \right\rfloor\right) & \text{otherwise.} \end{cases} \tag{3}$$

The three cases are explained as follows:

1. Case $k < s$:
   If the number of colors k is smaller than the minimum group size $s = 3$, it is impossible to form a valid group. Therefore, the group value is zero.

2. Case $\sum_{i=1}^{k-2} c_i \leq c_{k-1}$:
   In this case, the total number of copies from the first $k - 2$ colors is less than or equal to the number of copies of color $k - 1$. The optimal way to form groups is to pair each tile from colors 1 through $k - 2$ with one tile of color $k - 1$ and one tile of color k. This maximizes the number of valid groups by using all available tiles from the limiting colors (1 through $k - 2$), and since each of these tiles appears in exactly one group, no tile of these limiting colors remains unused. Therefore, this yields the maximum possible group value, equal to 3 times the number of tiles from the first $k - 2$ colors.

3. Case $\sum_{i=1}^{k-2} c_i > c_{k-1}$:
   In this case, the total number of tiles in the first $k - 2$ colors exceeds the number of tiles of color $k - 1$. As a result, we can no longer form all groups of size three by pairing colors $k - 1$ and k with a tile from one of the limiting colors, as we did in the previous case. Instead, we must determine how many complete groups we can form by pairing tiles from the first $k - 1$ colors with tiles from color k.

   Let $T = \sum_{i=1}^{k-1} c_i$ be the total number of tiles available from the first $k - 1$ colors. Since each group must consist of at least three tiles, and each groups requires exactly one tile of color k, the total number of groups we can form is bounded by both the number of color k tiles and the number of possible pairs within T. This yields:

$$n_{\text{groups}} = \min\left(c_k, \left\lfloor \frac{T}{2} \right\rfloor\right).$$

Note that c_k can be reduced to $\left\lfloor \frac{T}{2} \right\rfloor$ without changing the outcome of the group value formula, because any tile beyond this upper bound cannot be used in a valid group. Each group must include exactly one tile from color k and at least two tiles from the set of T tiles. If there are more than $\left\lfloor \frac{T}{2} \right\rfloor$ tiles of color k, the excess tiles cannot form valid groups because there would not be enough tiles left from T to satisfy the group size requirement. Therefore, any excess tiles of color k can be discarded. are redundant and can be discarded. The group value is thus given by:

$$T + \min\left(c_k, \left\lfloor \frac{T}{2} \right\rfloor\right)$$

After reducing c_k to $\left\lfloor \frac{T}{2} \right\rfloor$, valid groups can be formed without any leftover tiles, effectively making the group combination clearable. The process of group formation is as follows:

- Each group contains one tile from color k, which results in there being exactly n_{groups} groups.
- The minimal group size is $\left\lfloor \frac{T}{n_{\text{groups}}} \right\rfloor + 1$, with the additional tile coming form color k.
- Since T is not always divisible by n_{groups}, the remainder T mod n_{groups} is the number of groups with one additional tile.
- The groups are formed iteratively by taking the first color with the smallest number of tiles remaining and with the last $\left\lfloor \frac{T}{n_{groups}} \right\rfloor$ number of colors with the largest number of tiles remaining. For the first T mod n_{groups} we add the color previous to these last added colors too.

Figure 4 illustrates three examples corresponding to the last case, demonstrating how these groups are formed.

With the closed-form formula for the group value (Equation 3), the clearability check can be further simplified. To determine whether a row or the set of tiles with the same value is clearable, it suffices to verify whether the computed group value equals the total number of tiles in the row or same value set.

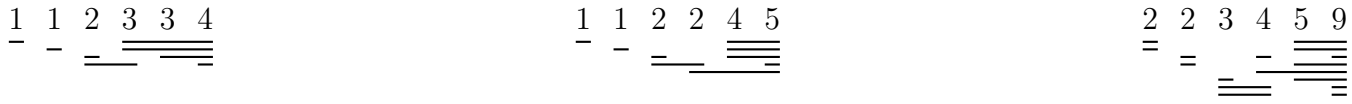

Figure 4: Three examples demonstrating how groups are formed when the sum of the first $k-2$ colors exceeds the number of tiles of color $k-1$. The horizontal bars indicate which tiles are used to form a group. In the third example, one tile of color k remains unused.

4.5 Group Value for Arbitrary Set Size

In order to extend the previous closed formula for fixed s, we generalize the logic used for fixed group sizes by introducing flexibility for arbitrary s. This formula builds upon the same principles of forming valid groups based on the number of available tiles, with additional consideration for excess tiles when s is no longer fixed to three. To compute the group value for arbitrary s, we adapt the previous formula by iteratively handling the excess tiles in the third case, ensuring that each group formed is valid according to the constraint of having at least s tiles while efficiently utilizing the available tiles from each color. This leads to the following formula:

$$Groupvalue = \begin{cases} 0 & \text{if } k < s, \\ s \times \sum_{i=1}^{k-(s-1)} c_i & \text{if } \sum_{i=1}^{k-(s-1)} c_i \leq c_{k-(s-2)}, \\ \sum_{i=1}^{k-(s-2)} c_i + \sum_{j=2}^{s-1} \min\left(c_{k-(s-j-1)}, \left\lfloor \frac{R_j}{j} \right\rfloor\right) & \text{otherwise.} \end{cases} \quad (4)$$

where $R_1 = \sum_{i=1}^{k-(s-2)} c_i$ and $R_j = R_{j-1} + \min\left(c_{k-(s-j-1))}, \left\lfloor \frac{R_{j-1}}{j} \right\rfloor\right)$ for $j \geq 2$.

The three cases are explained as follows:

1. Case $k < s$:
   In this case, it is impossible to form any valid group, since we need at least s distinct colors to construct a group. Therefore, the group value is zero.

2. Case $\sum_{i=1}^{k-(s-1)} c_i \leq c_{k-(s-2)}$:
   In this case, the total number of tiles from the first $k-(s-1)$ colors is less than or equal to the number of tiles of color $k-(s-2)$. The optimal way to form groups is to match each tile from colors 1 through $k-(s-1)$ with tiles from the remaining $s-1$ colors that follow. This guarantees that all the tiles from the limiting colors are used exactly once, maximizing the number of tiles used in valid groups. The resulting group value is s times the number of tiles from the first $k-(s-1)$ colors.

3. Case $\sum_{i=1}^{k-(s-1)} c_i > c_{k-(s-2)}$:
   In this case, the total number of tiles in the first $k-(s-1)$ colors exceeds the number of tiles of color $k-(s-2)$. As in the third case of the group value for fixed s, not all available tiles may be used to form valid groups. Since s is arbitrary and not fixed to three, we must iteratively handle the excess tiles across the last $s-1$ colors. We begin by forming groups of size three using the first $k-(s-2)$ colors. If not all tiles can be used, we remove excess tiles of color $k-(s-2)$, then increment the group size by one and include the next color $(k-(s-3)$. This process is repeated, each time increasing the group size and including the next color, until we reach group size s and use all the k colors. This approach ensures that the computed group value is the number of tiles used in valid groups by progressively removing excess ties that cannot be used.

Note that a formal proof that the group value formula for arbitrary s is optimal remains an open problem. However, for various values of s, k and m, we validated the formula using an automated script that exhaustively generates all valid group combinations for these parameters and compares their group value against the result produced by the formula. In all tested cases, the computed group value matched the formula's result.

5 Memory

In order to save previously known results and prevent the algorithm from calculating the same configurations, a state-encoding mechanism to optimize computations is introduced. This is achieved by two arrays called *streets* and *values*. The *streets* array keeps track of where formed runs end, the position of the last tile added to each run. By tracking these tiles, the algorithm can identify whether a given configuration has already been evaluated. Since the minimal set size is $s = 3$, we only consider the last four rows since it is the window in which runs from previous rows can end. More generally, for arbitrary minimal set size s we need to store the last $2s - 2$ rows to account for all the possible states a RUMMIKUB puzzle can have.
The encoding function takes all these values of ending runs in the current last four rows to represent a state of the current row. This encoded value is then used as an index in the *value* array. This array stores the maximum value computed for each unique state. Each entry is indexed by the current row number and the encoded state of the last four rows. If a state has already been computed the stored value is reused. Otherwise, the algorithm proceeds to compute the current state recursively and store the found value afterwards in the array. This ensures that each state is only computed once.

To compute the number of possible states in the last n rows, we exploit the similarity between groups and runs in the RUMMIKUB puzzle. Consider a column of length n filled with m copies of each tile. Runs can be formed from the bottom upward. When viewed from the top, the tiles in these formed runs are non-decreasing, which corresponds to the same group combinations discussed in Subsection 4.2. Figure 5 visually illustrate this similarity.

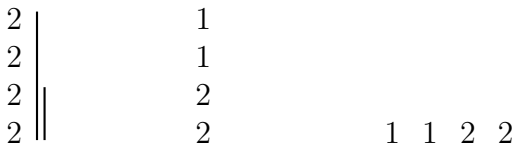

Figure 5: Visual representation of the similarity between group and run combinations. On the left, a column filled with 2 copies of each tile is shown. The formed runs are represented by the vertical bars to its right. In the middle, the tiles that are part of a run are shown. Viewed from the top this corresponds to the same non-decreasing group combinations to the right.

Due to this similarity, we can represent the configuration of the last n rows as a column-wise group combination. This allows us to alter Equation 1 to compute the number of unique states for these last n rows. Given parameters k and m, the total number of possible encoded states for the last n rows is:

$$\binom{m+n}{n}^k. \tag{5}$$

Since we encode the last 4 rows, we set $n = 4$. For $m = 2$, we get $\binom{2+4}{4} = 15$ unique possible states per column. Using $k = 4$ colors, this results in $15^4 = 50{,}625$ unique possible states for the last four rows. If we increase maximum the number of copies to $m = 3$, then there are $\binom{3+4}{4} = 35$ unique states per column, and with $k = 4$ we get $35^4 = 1{,}500{,}625$ unique possible states for the last four rows with four colors.

5.1 Possible Encoding Size

The number of unique states for the four last rows is now statically defined. However, not all these unique states can always be reached, as this depends on the initial configuration of the RUMMIKUB puzzle. For instance, if the last four rows in the initial configuration contain only zeros, there is only one unique state that can be reached: the last four zero rows. This leads to the following question:

> How many unique ways can runs be formed from the last four rows in one of the columns? In other words, how much memory does the algorithm require to encode the last four rows, given the initial configuration of the RUMMIKUB puzzle?

For $m = 2$, consider a column of length 4. We define two parameters to describe the structure of this column:

- Let w be the length of the longest contiguous sequence of non-zero elements starting from the bottom of the column.
- Let ℓ be the length of the longest continuous sequence of elements that are neither zero nor one (thus 2) starting from the bottom.

By definition $\ell \leq w$, since every 2 is also non-zero.
Note that:

- If the bottom value is 0, then both $w = 0$ and $\ell = 0$.
- If the bottom value is 1, then $\ell = 0$ and $w > 0$.

We define $P(w, \ell)$ as the number of possible unique encoded states in a column. The recurrence relation for $P(w, \ell)$ is given by:

$$P(w,\ell) = \begin{cases} \binom{w}{2} + w + w + 1 = (\frac{w}{2} + 1)(w + 1) & w = \ell, \\ P(w-1,\ell) + \ell + 1 & \text{if } \ell < w. \end{cases}$$

- **Base case** $w = \ell$: There are $\binom{w}{2}$ ways to choose two different run sizes. Since $w = 2$, we can also make two individual runs of length w. Finally, we include one additional case to account for the empty run.
- **Induction step** $\ell < w$: The sequence of 2s is shorter than the sequence of non-zero values. We can build upon all valid configurations from $P(w - 1, \ell)$ by adding a new non-zero (not 2) on top. This added tile introduces $\ell + 1$ additional configurations, because we can extend the sequence with 0 up to ℓ 2s at the bottom. Hence the added +1 indicating no 2s.

To derive a closed formula we solve the recurrence:

$$\begin{aligned} P(w,\ell) &= P(w-1,\ell) + \ell + 1 \\ P(w-1,\ell) &= P(w-2,\ell) + 2(\ell+1) \\ P(w-2,\ell) &= P(w-3,\ell) + 3(\ell+1). \end{aligned}$$

Eventually we get:

$$P(w, \ell) = P(\ell, \ell) + (w - \ell)(\ell + 1).$$

If we apply the base case:

$$\begin{aligned} P(w, \ell) &= \left(\frac{\ell}{2} + 1\right)(\ell + 1) + (w - \ell)(\ell + 1) \\ &= \left(\frac{\ell}{2} + 1 + w - \ell\right)(\ell + 1) \\ &= \left(w - \frac{\ell}{2} + 1\right)(\ell + 1). \end{aligned}$$

So, the closed-from formula for the number of states given a column is:

$$(w - \frac{1}{2}\ell + 1)(\ell + 1)$$

Figure 6 shows an example initial configuration of a RUMMIKUB puzzle. Using the closed-form formula derived above, we calculate the number of unique encoding for the last four rows per column:

- column b: $w = 0$ and $\ell = 0$ results in one unique state.
- column g: $w = 4$ and $\ell = 1$ results in 5 unique states.
- column r: $w = 4$ and $\ell = 2$ results in 12 unique states.
- column y, $w = 4$ and $\ell = 4$ results 15 unique states.

Multiplying these number of unique states per column gives the total number of possible states for the last four rows:

$$1 \times 5 \times 12 \times 15 = 900$$

This is significantly fewer than the previously stated number for $m = 2$, which was $15^4 = 50{,}625$. This example shows that a dynamic memory allocation approach can significantly reduce the total memory usages required to solve a RUMMIKUB puzzle. Instead of storing values for all possible states of the last four rows, the algorithm only needs to track all the reachable states from a given initial configuration.

Row	b	g	r	y
5	0	1	1	2
6	2	1	1	2
7	1	1	2	2
8	0	1	2	2

Figure 6: Example last four rows in the initial configuration of a RUMMIKUB puzzle.

6 Puzzle Bounds and Reduction

Before assessing the difficulty of RUMMIKUB puzzles, we define the puzzle bounds and apply reductions to simplify the problem.

6.1 Puzzle Bounds

In order to categorize the difficulty of different RUMMIKUB puzzles, we calculate both a lower and upper bound for *maxvalue*, the maximal value that can be reached. A straightforward lower bound can be obtained by forming only runs or only groups. The value obtained by forming only runs is called the *run value* Os and the value found by only forming groups *group value* Og. Since the number of points of a RUMMIKUB puzzle cannot be less than either of these values, the lower bound is therefore: $\max(Os, Og)$.

For the upper bound, an intuitive approach is to sum all the values on the tiles, which we refer to as *totalpoints*. This represents the maximum score achievable, assuming all tiles could be used in a valid group or run. However, some tiles may not be usable in any groups or runs, so this upper bound might not always be tight. To sharpen the upper bound, we take the sum of the points from only runs and from only groups, and we define the upper bound as: $\min(Os+Og,\ totalpoints)$.

Thus, the following holds: $\max(Os, Og) \leq maxvalue \leq \min(Os + Og, totalpoints)$.

If the lower bound of a RUMMIKUB puzzle equals the upper bound, then the puzzle is already solved. all tiles can be played by forming either only groups or only runs, thereby achieving the maximum points possible. In this case, we can stop the algorithm early, as the solution is already known.

6.2 Run Value

In order to find the run value of a series of tiles of the same color, given $m = 2$, we proceed as follows. We iterate over all the same colored tiles, zero copies are excluded, as they cannot contribute to a run. We only consider runs with length ≥ 3, since minimal run length $s = 3$. The run value is the sum of all 1s and 2s, adjusted with their tile values, with the following exceptions:

- A singleton 2 must be preceded and succeeded by at least two non-zero elements. If not, one of the two tiles is excluded from the sum.
- Two consecutive 2's must be preceded and succeeded by at least one non-zero element. If not, again one of the two consecutive tiles is excluded from the sum.

To generalize these rules for $m = 3$, we extend the previous exceptions as follows:

- A singleton 3 must be preceded and succeeded by at least two elements. There are two distinct cases:
  - The singleton 3 is at least preceded by two 2's and succeeded by two 1's or vice versa.

– the preceding and succeeding elements are 2's and the elements on both ends are non-zero.

- Two consecutive 3's must be preceded and succeeded by at least one non-zero element. One of the neighboring tiles must be at least a 2, while the other may be a 1.

The rules described above seem to have a pattern that even holds for $m > 3$. We define t with $0 < t < m$ as a variable number of copies found in a RUMMIKUB puzzle.

For a singleton t two non-zero elements should precede and succeed this singleton number of copies. The total value of these four elements that precede and succeed our singleton element should add up to at least two times the singleton number of copies, with the constraint that the closest preceding and succeeding elements are greater than or equal to the corresponding furthest proceeding and succeeding elements.
For the two consecutive number of copies t, the following rule applies. The two consecutive number of copies should be preceded and succeeded by at least one non-zero element and the sum of these two elements should be at least t.

For both the singleton and consecutive t cases, if the rule does not apply to the current configuration, one of the t number of copies tiles should be excluded from the run value and the rules should be reapplied continuously until the right number of copies that are included in the run value is found.

Figure 7 shows the minimum configurations for singleton and consecutive t's, for $t = 3$ and $t = 4$. The t number of copy tiles in the middle are all included in the run value in these cases. The non-t-values can be raised without affecting the validity of the run. However, when lowered, the rules no longer apply, resulting in a tile in t being excluded. Additionally, these configurations can be flipped (top to bottom), and the rules will still hold.

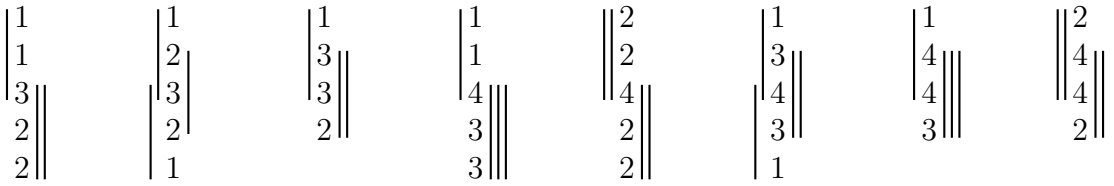

Figure 7: All the minimal configurations for $t = 3$ (on the left) and $t = 3$ (on the right), for all the t number of tiles to be included in the run value. The vertical bars on the side indicate the valid formed runs for these configurations.

6.3 Pre-processing

The upper bounds described in Section 6.1 can be even sharper. The value *totalpoints* calculated by taking the sum of all the tiles present in the RUMMIKUB puzzle. However, some tiles can be filtered out. In particular, certain isolated tiles can be eliminated during pre-processing. These tiles cannot be part of a valid run or group. When a tile cannot appear in a possible group or run, this tile cannot be part of the solution to the RUMMIKUB puzzle. Removing these isolated tiles will sharpen the upper bound, since there are now fewer tiles in the total number of points and it will reduce the problem size of the RUMMIKUB puzzle.

In Figure 8, an example RUMMIKUB puzzle configuration with isolated tiles is shown. In row 3 and column y there are two copies of this tile. However, since there is no possibility to form a group or a run, these tiles are isolated and therefore cannot be part of a valid solution to this RUMMIKUB puzzle.

Row	b	g	r	y
1	1	0	**2**	1
2	1	2	**2**	0
3	0	1	0	**2**
4	2	2	2	0
5	2	1	2	1
6	1	2	2	2

Figure 8: An example RUMMIKUB puzzle configuration with isolated tiles. In row 3 and column y, two copies of this tile are shown, but these tiles cannot be part of a group or run, which makes them isolated. In the first and second row we see 2 copies in column r. In both rows only one group and no runs can be formed. Therefore one of these two copies is isolated and cannot be part of a group or run.

Algorithm 2 Algorithm that detects and removes isolated tiles in a RUMMIKUB configuration with $k = 4$ and $m = 2$.

Require: RUMMIKUB configuration $G(k, m, n)$ represented as a 2D array `puzzle` with n rows and k columns (colors).
Ensure: Returns the number of isolated tiles removed and updates the configuration in-place.

```
Initialize counter isolated_tiles to 0
for each row from the last row until the first do
    if row is not clearable then
        for each column in the row do
            if (groupvalue > 0 and copies > 1) or copies ≠ 0 then      ▷ Potential isolated tiles
                runs = possible runs with [2 tiles below, 1 above & below, 2 tiles above]
                potential_isolated = copies − runs
                if groupvalue > 0 then                    ▷ 1 of the copies can be used in the group
                    potential_isolated − = 1
                end if
                if potential_isolated > 0 then                                    ▷ isolated tiles
                    isolated_tiles + = potential_isolated
                    remove the isolated tiles in this row and column.
                end if
            end if
        end for
    end if
end for
return isolated_tiles
```

6.4 Splitting Rummikub Puzzles

In order to reduce the problem size of the RUMMIKUB puzzle, some initial configurations were found that could reduce the RUMMIKUB puzzle into smaller RUMMIKUB puzzles. A straightforward example is a row of zeros, as shown in Figure 9. The zeros row act as an barrier between the rows above and below, preventing the possibility of forming runs between these rows. This separation allows the puzzle to be split into smaller independent puzzles. By splitting the RUMMIKUB puzzle, we reduce the number of permutations that need to be calculated, thereby decreasing the overall problem size. Therefore, identifying configurations where the puzzle can be split into smaller puzzles is beneficial for solving the puzzle efficiently.
Another configuration that allows splitting is when two rows each contain at least one zero in ever column. As illustrated in Figure 10, these rows cannot form runs with each other, so they can be split and treated like separate puzzles.
However, for three rows where each column contains at least one zero, the puzzle cannot be split in the same way, as the split eliminates possible runs between the columns.

Row	b	g	r	y
1	1	0	1	1
2	1	2	0	2
3	0	1	1	2
4	**0**	**0**	**0**	**0**
5	0	2	1	2
6	1	2	2	2
7	2	2	2	2

(a) Initial configuration.

Row	b	g	r	y
1	1	0	1	1
2	1	2	0	2
3	0	1	1	2

5	0	2	1	2
6	1	2	2	2
7	2	2	2	2

(b) Zeros row cut in two smaller RUMMIKUB puzzles.

Figure 9: Example RUMMIKUB configuration, which can be split into smaller RUMMIKUB puzzles by cutting the zero row.

Row	b	g	r	y
1	1	2	1	1
2	1	2	2	0
3	**0**	1	**0**	1
4	2	**0**	1	**0**
5	1	2	1	2
6	1	1	1	1
7	0	0	0	1

(a) Initial configuration.

Row	b	g	r	y
1	1	2	1	1
2	1	2	2	0
3	**0**	1	**0**	1

4	2	**0**	1	**0**
5	1	2	1	2
6	1	1	1	1
7	0	0	0	1

(b) Zeros rows cut in two smaller RUMMIKUB puzzles.

Figure 10: Example RUMMIKUB configuration, which can be split into smaller RUMMIKUB puzzles, by splitting between the two rows in which there is at least one zero in each column.

7 Difficulty Classification Rummikub Puzzles

In this section, we attempt to classify the difficulty of RUMMIKUB puzzles using only precomputed puzzle features derived from the initial puzzle configuration. The goal is to predict the difficulty of a puzzle without executing the solving algorithm. In this context, the difficulty of a puzzle is defined by the execution time required by the solving algorithm. While execution time is not a theoretical measure of difficulty, it serves as a practical approximation of the computational effort needed to solve the puzzle. Puzzles with longer execution times typically involve more complex interactions between forming groups and runs.

7.1 Exploration

To better understand which types of RUMMIKUB puzzles are computationally challenging, we conducted an exploratory experiment across a range of puzzle configurations. We generated 100 random RUMMIKUB puzzles with $m = 2$ and $n = 100$, each based on a fixed distribution of tiles with zero, one or two copies. The experiment systematically varied the percentages of tiles with zero and one copies, with the percentage of two copies tiles determined as the remaining proportion. For each such combination, 100 puzzles were generated and their average execution time was recorded. The results are shown in Figure 11. A small region stands out where the average execution time is significantly higher than in other areas. Intuitively, puzzles with a high percentage of zero-copy tiles are computationally inexpensive to solve, as few or no valid runs or groups can be formed with only zero or one copy per tile. In contrast, puzzles with low percentages of zero and one copy tiles or alternatively high proportion of two copy tiles are more computationally demanding. Having two copies of many tiles significantly increases the number of possible group and run combinations, which leads to longer execution times.

In Figure 12, we zoom in on the most computationally expensive region of the experiment, specifically the 0–10% range. We observe that the average execution time is significantly lower when the percentage of two copy tiles approaches 100%. In other words, when almost every tile is available, the puzzle becomes easier to solve. This is because when each tile appears twice, the puzzle can be easily solved by forming only groups, as most rows become clearable with a high proportion of two copy tiles. Thus, the most computationally expensive puzzles have a high proportion of two copy tiles, but not a full set of two copies, as a complete set would make the puzzle solvable through groups alone.

7.2 Feature Selection

The complexity of a RUMMIKUB puzzle is intuitively related to the number of ways in which valid groups and runs can be formed. To capture this notion of difficulty, we selected features that are expected to correlate with the number of possible combinations of groups and runs the algorithm must explore during execution.
The selected features are:

- **Upper–lower bound difference**: An upper and a lower bound can be easily computed, and their difference may indicate the puzzles complexity. A zero difference implies that the puzzle

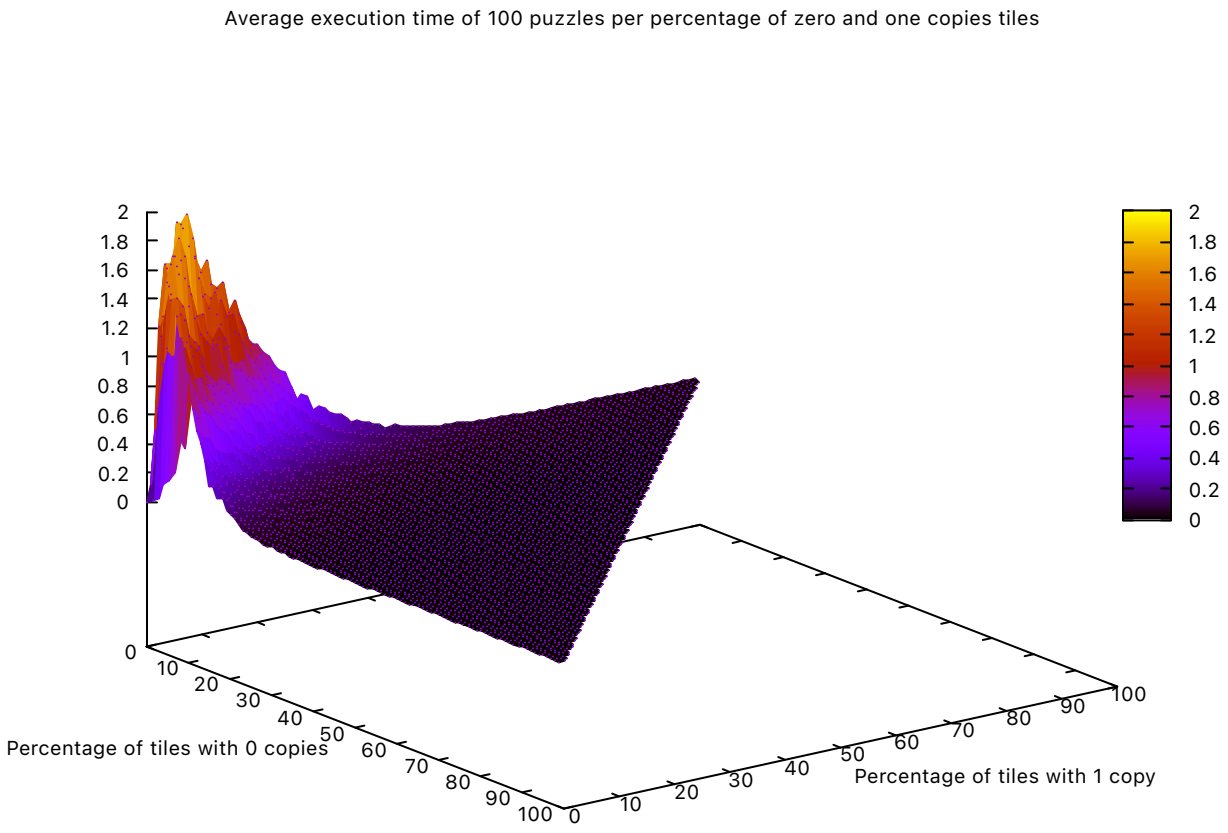

Figure 11: Average execution time over 100 random RUMMIKUB puzzles, plotted for varying percentages of zero and one copy tiles (0–100%).

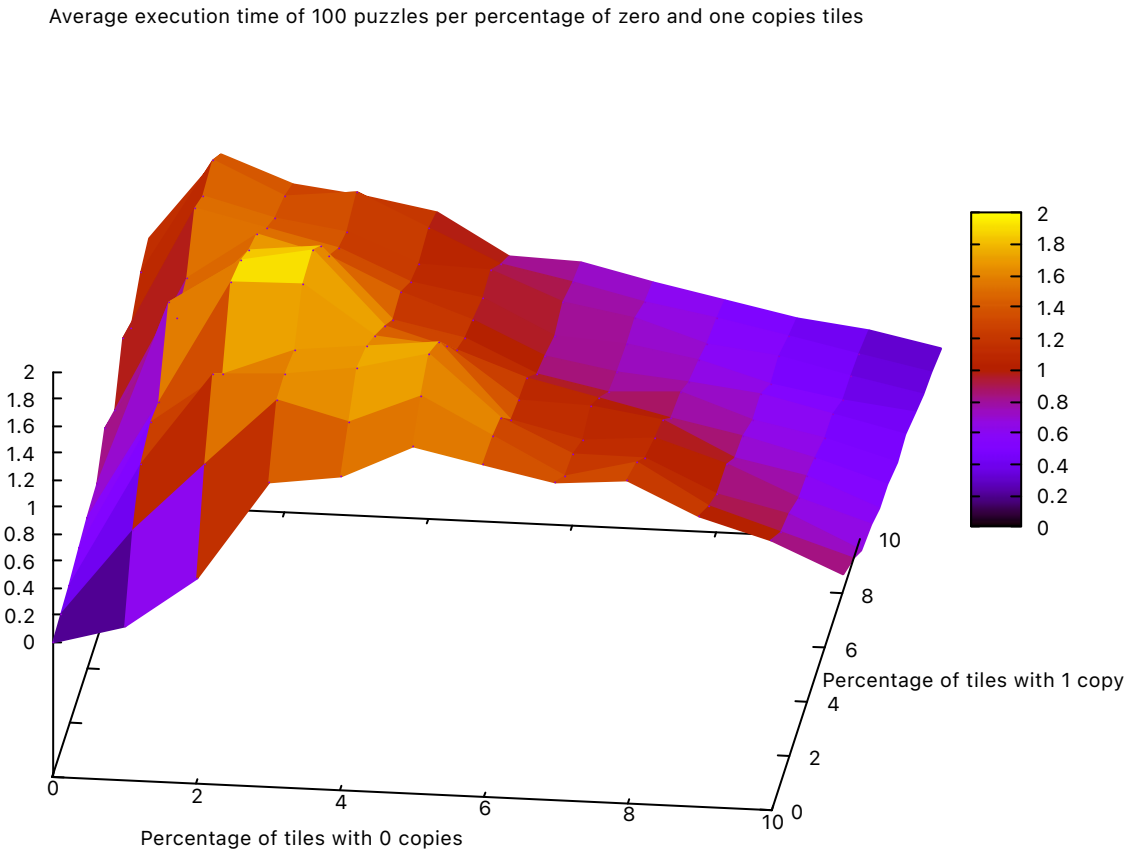

Figure 12: Zoomed-in view of Figure 11, zooming in on the percentage range of 0–10% of zero and one copy tiles to highlight the computationally expensive region.

can be solved by forming only groups or only runs, while a larger difference may imply more complex interactions between groups and runs, which may indicate a more difficult puzzle.

- **Tile count difference (upper-lower bound)** The difference between upper and lower bound includes the tile values, which can introduce a bias. A high-valued tile that is excluded contributes more to the difference than a low-valued excluded tile. To correct for this, we normalize the difference by considering only the number of used tiles in each bound.
- **Percentage of tiles with 0, 1 and 2 copies**: A higher proportion of tiles with two copies increases the number of possible runs and groups. In contrast, a higher ratio of zero-copy tiles reduces the number of valid combinations. Note that puzzles that have two copies for every tile are easy to solve by only forming groups. This suggests a trade-off between the percentages of the number of copies of tiles.
- **Tiles usable in both runs and groups**: Tiles that can be used in both a group and a run introduce additional branching, increasing the number of permutations the algorithm must evaluate. A puzzle with a high proportion of such tiles may indicate an increased difficulty.

7.3 Training Data

The model is trained on a dataset consisting of 400,000 unique RUMMIKUB puzzles generated from the configuration $G(4, 2, 100)$. Each puzzle in this dataset has precomputed features derived from its initial configuration to determine the difficulty of the puzzle.

7.4 Difficulty Labeling

While the execution time is not directly used as a feature during the training of the model, it is used to define the difficulty classes for supervised learning. Since it is currently the only indication of a RUMMIKUB puzzle complexity. Based on the observed execution time, we define the following difficulty levels:

- **1 (Easy)**: Execution time < 0.0001 seconds.
- **2 (Moderate)**: $0.0001 \leq$ Execution time < 0.1 seconds.
- **3 (Medium)**: $0.1 \leq$ Execution time < 1 seconds.
- **4 (Hard)**: Execution time ≥ 1 seconds.

The distribution of puzzles across these difficulty classes is shown in Table 3.

7.5 Model Selection

For the task of classifying RUMMIKUB puzzle difficulty, we selected the Random Forest Classifier as our model, originally introduced by Breiman [Bre01]. Random forest is an ensemble learning method that constructs multiple decision trees during training and aggregates their outputs through majority voting. This model was chosen for its robustness to overfitting, suitability for imbalanced data and feature importance analysis. The feature importance is the most interesting, as we aim to identify which features are the most predictive of a puzzle's difficulty.

Difficulty class	Number of puzzles	Percentage of total
Easy	103,636	25.9%
Moderate	69,425	17.4%
Medium	112,179	28.0%
Hard	114,760	28.7%

Table 3: Distribution of RUMMIKUB puzzles across the four difficulty classes.

7.6 Model Training and Evaluation

The model training process involves the following steps:

- **Hyperparameter Tuning:** We performed hyperparameter tuning using grid search with cross-validation to optimize the performance of the Random Forest classifier. The best-performing parameters were:
  `n_estimators=150`, `min_samples_split=10` and `min_samples_leaf=4`.
- **Cross-validation:** We used 10-fold cross-validation to evaluate the model's performance. This approach ensures that each puzzle appears in both the training and validation sets across different folds.
- **Evaluation Metrics:** We used classification accuracy, confusion matrix and the classification report (precision, recall and F1-score) to asses the model's performance across all four difficulty classes.
- **Feature Importance:** To gain insight into which features of a puzzle are most predictive of its difficulty, we extracted the feature importance from the final Random Forest model trained on the entire dataset.

7.7 Results

Table 4 shows the classification report for the model. We can observe that the model performs well in predicting puzzles in the easy and moderate difficulty classes, with high precision and recall scores. However, its performance is significantly lower for the medium and hard classes. This is interesting, as the moderate class is underrepresented compared to the medium and hard classes in the dataset. In the confusion matrix shown in Figure 14 we observe that medium and hard puzzles are often confused with each other. This suggest that distinguishing between puzzles becomes more challenging as the difficulty increases. In Figure 13, the ranking of feature importance is shown. The number of tiles that can be part of both a group and a run is shown to be the most predictive indicator of puzzle difficulty. Notably, the difference in bounds ranks higher than its normalized counterpart, the difference in the number of tiles in the upper and lower bounds.

8 Conclusion and Future Work

Throughout this thesis, we focused on efficiently solving RUMMIKUB puzzles by addressing both the optimization of the algorithm's computational performance and the categorization of puzzle

Class	Precision	Recall	F1-Score	Support
Easy	0.97	0.94	0.95	103,636
Moderate	0.92	0.93	0.92	69,425
Medium	0.72	0.74	0.73	112,179
Hard	0.74	0.74	0.74	114,760
Accuracy	0.82 (400,000 total)			
Macro avg	0.84	0.84	0.84	400,000
Weighted avg	0.83	0.82	0.82	400,000

Table 4: Classification report for RUMMIKUB puzzle difficulty classification.

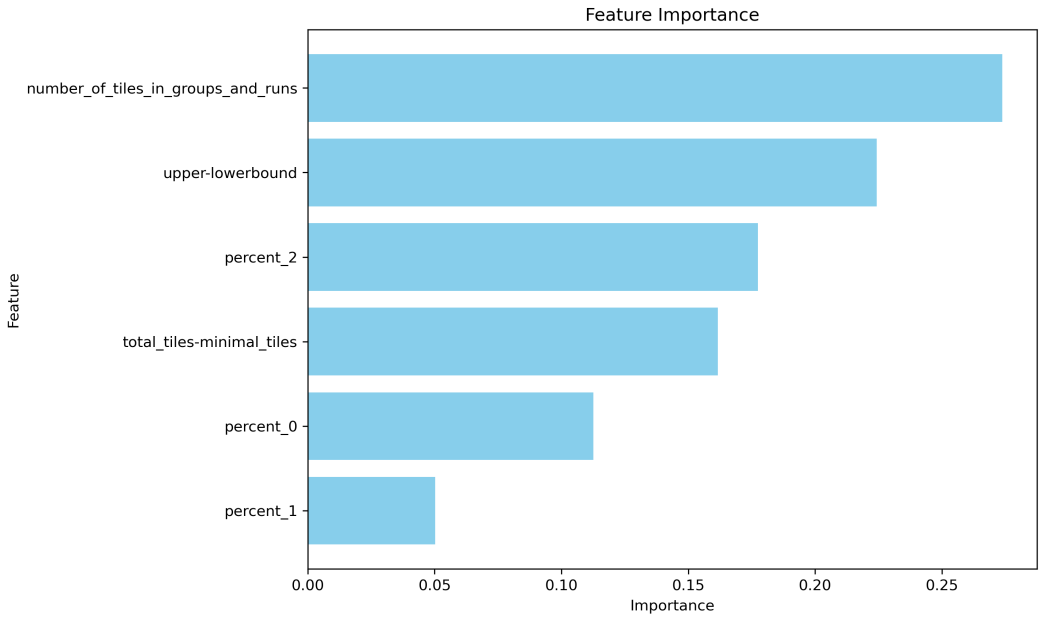

Figure 13: Feature importance plot for RUMMIKUB puzzle difficulty classification.

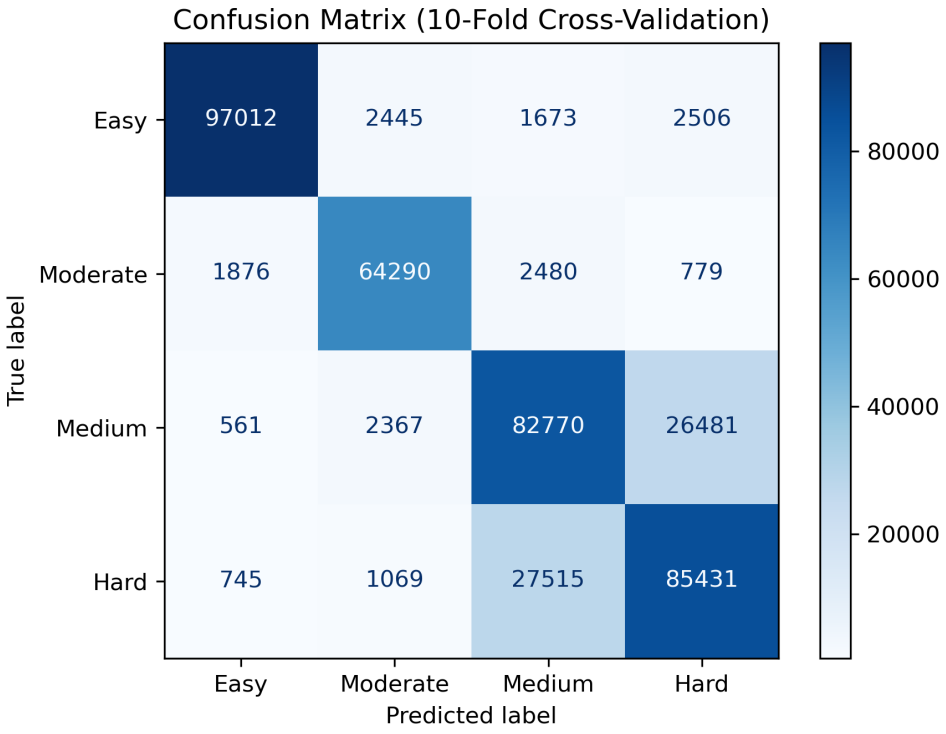

Figure 14: Confusion matrix for RUMMIKUB puzzle difficulty classification with 10-fold cross-validation.

difficulty. To optimize the computational performance, we introduced a state-encoding mechanism that prevents redundant computations. We further extended this approach with dynamic memory allocation, ensuring that memory is only allocated for states that are reachable from the initial configuration. Additionally, we reduced the problem size by analyzing puzzle configurations that could be broken down into smaller puzzles.

To categorization puzzle difficulty, we used execution time as an indirect indicator to define difficulty levels. While it does not perfectly capture the underlying complexity, it offers a practical way to distinguish between easier and more complex puzzles. As the puzzle difficulty increases, classification becomes more challenging. Among the feature analyzed, the number of tiles that can be part of both groups and runs proved to be the most predictive of a puzzle's difficulty.

In this thesis, we primarily focused on solving Rummikub puzzles with varying parameters such as the number of tile copies, the number of colors/suits and the number of tile values. An interesting extension would be to add joker tiles, which can be substituted for any other tile. This introduces additional complexity to forming valid groups and runs, and raises the question on the value of jokers in the Rummikub puzzle.

Another promising direction for future research is to translate the findings of this thesis to the multiplayer Rummikub game. Little research has been conducted on this multiplayer variant, which introduces several new concepts like: turn based-dynamics, table constraint, initial meld, manipulations of existing sets and strategy. Exploring these aspects will move the problem closer to the original game, and open research for strategy with Artificial Intelligence.

It would be interesting to further explore the difficulty categorization of Rummikub puzzles, as there may be features not examined in this thesis that could provide better insight into the differences between more difficult puzzles. Additionally, we could use the memory usage of Rummikub puzzles instead of execution time. This metric could be more closely related to puzzle difficulty, as it reflects the number of possible combinations of groups and runs that need to be explored to solve the puzzle.

Proving the group value for arbitrary minimal group sizes remains an open problem. If the group value for arbitrary s can be settled, it would allow us to compute Rummikub puzzles in a more generalized manner, since runs are easily adaptable to arbitrary minimal set sizes.

References

[Bre01] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[dHH06] D. den Hertog and P. B. Hulshof. Solving Rummikub Problems by Integer Linear Programming. *The Computer Journal*, 49(6):665–669, 2006.

[Fei23] R. Feijen. Counting Winning Hands in Rummikub. Bachelor Thesis, LIACS, Leiden University, 2023.

[Gul19] M. Gulin. Solving Rummikub with Computational Power. Thesis, Åbo Akademi Univeristy, 2019.

[KK03] C. Kotnik and J. Kalita. The Significance of Temporal-Difference Learning in Self-Play Training TD-Rummy versus EVO-rummy. In *Proceedings of the 20th International Conference on Machine Learning*, pages 369–375, 2003.

[Lem] Lemada Light Industries ltd. Rummikub Classic Rules. https://rummikub.com/rules/. Accessed: 26-03-2025.

[VMLV25] S. Vandevelde, L. Mertens, S. Lauwers, and J. Vennekens. Enhancing Computer Vision with Knowledge: a Rummikub Case Study. In *Proceedings of ESANN 2025*, pages 201–206, 2025.

[vRTV15] J. N. van Rijn, F. W. Takes, and J. K. Vis. The Complexity of Rummikub Problems. In *Proceedings of the 27th Benelux Conference on Artificial Intelligence (BNAIC 2015)*, 2015.