Counting winning hands in Rummikub

Romke Feijen

Supervisors:
Jan N. van Rijn & Jonathan K. Vis

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 01/05/2023

Abstract

Rummikub is a tile-based game, where players in turn play tiles from their hand to the table. Tiles on the table must be grouped by value with different colors or grouped by color as a consecutive range of values. The first player to get rid of all their tiles wins. This thesis will focus on the possibility that given an arbitrary set of tiles, that set is winning. Such knowledge is relevant when developing game strategies for example. We will describe an algorithm that will count the number of winning combinations that can be formed with this given set. The algorithm produces results that correspond with other results from different work and creates confidence that these results are accurate. Using the full default tile set of Rummikub makes the problem too large for current hardware to handle, thus a smaller set of tiles is used to test the capabilities of this algorithm and find its limit. For our implementation, the bottleneck seems to be memory usage. Hence a suggestion for future work is to improve the efficiency of the algorithm's memory usage and two concrete pointers are given.

Contents

1 Introduction

Rummikub is a tile-based game, played with a set of 106 (104 + 2 jokers) tiles. A tile has a value range from 1 to 13, one of four colors, and one other copy of itself. Figure 1 shows the full tile set for Rummikub. Each player starts with 14 tiles and at every turn, they can decide to play a set of tiles or draw a new one. A player can play its tiles by forming groups with these tiles either by grouping by value with different colors or grouping by color in consecutive number order. The first player with no tiles left wins the game.

A player's chance of winning is very valuable information at any point in a Rummikub game, as this could influence a player's decision-making at that point. Such information could potentially lead to developing strategies to give the player a higher chance of winning.

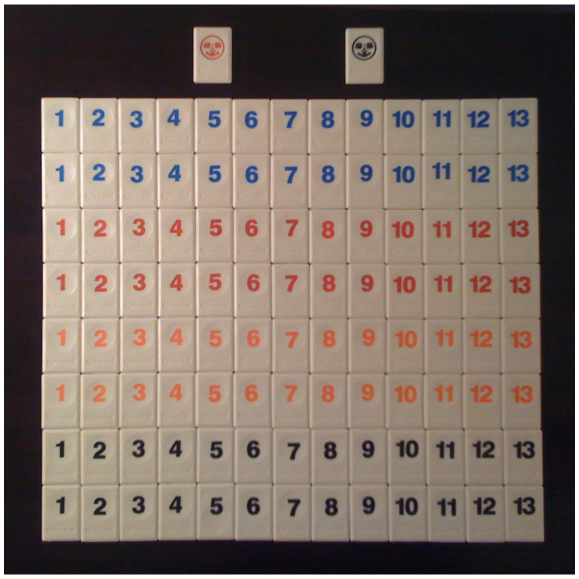


Figure 1: The default Rummikub tile set

In a highly relevant study, van Rijn et al., 2015 aimed to answer the hypothetical question "Given a hand of h tiles, what is the probability that hand is winning?". The paper explains the rough outlines of an algorithm and follows up with two series of results. One is with the default Rummikub tile set (13 values, 4 colors, and 2 copies of each), which shows to be a challenge to compute completely. The second was for a custom Rummikub tiles set (6 values, 4 colors, and 2 copies of each) for which all solutions were found.

Their work opens some opportunities for further research and explanation. The algorithm, for one, can be described in more detail and the results achieved have yet to be confirmed to be correct. A lot more results can be gathered with variations on the tile set as well, not just by changing the value parameter, but also the color or copy parameters. Finally, there is a clear indication that for some tile sets some limits are hit in terms of performance and hardware, but it is not clear where these limits lie and what that means on current hardware.

This thesis aims to continue where van Rijn et al., 2015 left on the question “Given a hand of h tiles, what is the probability that hand is winning?” and attempt to answer other questions mentioned above. We developed an algorithm and explain its exact working. Another approach is briefly explored and it is discussed why this approach is not ideal. The developed algorithm is then put to the test to check if we can confirm the counting results from the original paper, but also how the results take shape for other parameters, both in terms of counting sizes and runtime. We will investigate what tile sizes we compute completely with current hardware.

To summarise, this thesis will explore the computational limits of algorithms that count winning Rummikub hands. Section 2 will explain the rules of Rummikub, as well as how these rules are perceived and applied in this research. Section 3 will discuss some related work that is already done on this subject or is relevant in another way. The methodology of exactly how the problem has been tackled and how the algorithms work is stepped through in section 4. The implementation and further optimizations are covered in section 5. The performance and the results of the implementation of this algorithm are discussed in section 7. Section 8 will round up this thesis and go into some potential further research.

2 Definitions

This section 2.1 will give a summary of the classic "Rummikub: How to play", n.d. rules. For this thesis we will make an adoption on this set of this rules to both keep the complexity of the problem manageable as well as confine the research. This adoption of the rules is described in section 2.2. Finally, section 2.3 will explain some of the notable parameters and terminology around describing and representing Rummikub scenarios.

2.1 Classic rules

Rummikub is a tile-based game that is played with one or more players. The tile set consists of 13 values $\times$ 4 colors $\times$ 2 copies of tiles, where each **tile** has a value and a color. The tile set can contain extra tiles, like jokers, which can take up any color and value. A game of Rummikub starts with 14 tiles in the **hand** of each player. Players take turns to either draw a new tile from the undistributed tile pool or play their tiles by placing them on the **table** for everyone to see. Tiles can only be played in a **set** such that all tiles on the table are either formed in a **group** or a **run** with a minimal size of 3. Groups are formed with tiles of the same value but each is a different color from the other. Runs of tiles have the same color but are always in a contiguous and consecutive order. The tiles placed on the table can be rearranged by whichever player's turn it is, as long as all the tiles placed on the table are in valid runs and groups by the end of the turn. Once tiles are played on the table they can no longer be picked up from the table.

Players can choose to enforce an initial meld, where the very first play a player makes must at least add up to 30 points. The goal is for all players to keep playing in turns until the player has no more tiles in their hand. The first player to play all their tiles validly will win the game.

2.2 Generalisations for this research

For this thesis, there are a few generalizations made to the default rule set. The motivation behind these changes is mostly to keep the problem manageable and keep more focus on the counting problem at base.

Firstly, we are only interested in the hands of a player. Considering the hands of adversaries or mechanics like taking turns are not of interest here. By playing Rummikub alone, these aspects of the game are not relevant and can be disregarded, therefore for the rest of the paper, we are always considering a single-player version of Rummikub. As a single-player game, there is no real need to distinguish tiles on the table from the tiles in the player's hand, therefore it will be assumed that all tiles are always in the player's hands. This hand can be considered to be **winning hand** if it is possible to arrange all tiles in the players hand in such a way that each tile is in a valid set.

The requirement for an initial meld is left out, as it would filter out a lot of lower-valued hand sizes that are as interesting in the perspective of counting and totality. Jokers are also not in the game, due to their added complexity.

Requirements around tile placement and set formation will remain in place but could be tuned. The next section will go through some of the parameters that can be distinguished.

2.3 Terminology

Throughout this thesis and several parameters will be used. Most notable will be **hand size** h, the number of tiles kept in a hand. The exact value or color of these tiles is unknown. Three other important (input) parameters for this thesis are n, k, and m:

- n: the maximum value a tile can take
- k: the number of different colors a tile can take
- m: the number of copies that exist of a tile

Together $n \times k \times m$ also binds the maximum value of h. Furthermore, they respectively form the *rows* $\times$ *columns* $\times$ *values* of a matrix that can represent a **configuration** of tiles in a table. This form of representation will be used throughout this thesis to explain several principles. Figure 2 show an example of such a representation.

0	0	1	0
1	1	2	0
0	0	1	0

Figure 2: 3D visualisation for the Rummikub playing field $n = 4, k = 3, m = 2, h = 6$.

Here, runs are placed on the rows and groups on the columns. The value in each cell represents the number of copies for that tile. Note that this situation in the figure is winning because it consists of a run on the 2^{nd} row of 123 and a group on the 3^{rd} column with 3's.

3 Related Work

The amount of research on the game Rummikub remains small, with only a handful of papers directly related to the game itself.

Some of the earlier work is done by den Hertog and Hulshof, 2006. Here the authors challenge themselves with the question "Given a set of tiles, what is the maximum number or value of tiles that can be played?". They show this can be solved using a mathematical model: integer linear programming. They extend this model by minimizing the changes done to already existing sets.

More recently, Gulin, 2019 supposedly attempts to solve Rummikub using two types of algorithms: a recursive method and a heuristically guided space search. Due to restricted access to the paper itself, the work cannot be further researched for the writing of this paper.

This paper is mostly a continuation of the work by van Rijn et al., 2015. In this paper the authors consider the problem of "Can a set of given tiles be placed in such a way that each tile is placed in a valid run or group?" as an optimization problem. They show that this problem can be solved in polynomial time and as part of that solution provide an algorithm that works with different input variables. The same algorithm was then also used for the counting problem: "Given a subset of tiles, what is the chance this hand can be played in one move?". Continuing on this very subject, the goal is to confirm the results of these authors, push for higher input sizes, consider different implementations in practice, and discuss the performance of these implementations.

Less related but still relevant is the work by Kotnik and Kalita, 2003 where Temporal-Difference Learning is compared against an evolutionary algorithm with Rummy, a card variant of Rummikub.

4 Methods

Given a hand with a number of tiles, of which the values and colors are unknown, there are a finite number of different configurations of tiles that can be made for this hand. The work by van Rijn et al., 2015 shows us that the ratio between the total configurations and the configurations that are of such different proportions it would naive and infeasible to consider every possible configuration to find the ones that are winning. While section 6 and 7.2 will show that there are some specific scenarios where this might work, it will also show that for most hands sizes it will not work. Hence a different approach is suggested in this section. Counting Winning (Rummikub) Hands (**CWH**) is an algorithm that systematically enumerates all the winning configurations based on the rules of the game.

4.1 The algorithm

The concept of **enumerating** configuration, could be thought of as repeatedly adding and removing tiles to a hand to form all the different variations that exist. The core principle of CWH is that it should only ever enumerate winning configurations, even during the operation of the algorithm. This means when enumerating configurations for a given number of tiles, for every tile added to the configuration, the hand should remain winning regardless of whether the target size has been reached or not. To guarantee this, the algorithm is bound to the rules of Rummikub, the most important one being: *tiles may only be added to the configuration in groups or runs with a minimal size of 3*. Consequently, it will not even be necessary to ever add more than 5 tiles at once. From then on any number can be partitioned by 3,4 and 5, meaning that there exists a combination of sets of length 3, 4, and 5 such that they all add up to that number.

Next, CWH will be explained. Figure 3 will function as simple visualization of this algorithm where unique tiles exist (no copies, $m = 1$). The pseudo-code might also help as a guideline. It can be found in algorithm 1 and is explained in section 4.1.1.

$$\begin{array}{|c|c|c|c|}\hline 1&1&1&0\\\hline 0&0&0&0\\\hline 0&0&0&0\\\hline\end{array}\ ,\ \begin{array}{|c|c|c|c|}\hline 0&1&1&1\\\hline 0&0&0&0\\\hline 0&0&0&0\\\hline\end{array}\ , \dots ,\ \begin{array}{|c|c|c|c|}\hline 0&0&0&0\\\hline 0&0&0&0\\\hline 0&1&1&1\\\hline\end{array}\ , \dots ,\ \begin{array}{|c|c|c|c|}\hline 1&0&0&0\\\hline 1&0&0&0\\\hline 1&0&0&0\\\hline\end{array}\ , \dots ,$$

$$\begin{array}{|c|c|c|c|}\hline 0&0&0&1\\\hline 0&0&0&1\\\hline 0&0&0&1\\\hline\end{array}$$

Figure 3: An example of an iteration of the CWH algorithm for a simplified configuration: $n = 4, k = 3, m = 1, h = 3$.

CWH will consider a configuration as a matrix, over which it can iterate from left to right, top to bottom. By iteratively adding runs and groups to this configuration a solution can be built. Placing the set slightly differently each iteration, as seen in figure 3, will create a new solution every time. In essence, this is how CWH operates.

The algorithm will do two full passes over all positions in the matrix. On its first pass of the matrix, only runs will be placed. In every iteration, CWH will evaluate its options at that current position and the number of tiles remaining in its hand. For example, a run placement may never exceed the maximum value n. Similarly, if at any point there are only 4 tiles left to play, these tiles have to be played all at once. Playing only 3 of the 4 tiles would leave 1 tile remaining, which is against the design philosophy of this algorithm. Choosing not to place a run is also an option. Placing a run will initiate a recursive call, passing its state as well and the number of remaining tiles with the number of placed tiles subtracted.

Once the end of the matrix has been reached and all the run placement options have been exhausted but CWH still has remaining tiles to play, it will consider group forming. This is a slightly different approach because where runs are in consecutive numerical order, groups are unordered. As opposed to the runs placement, group forming will be done on the vertical axis of the matrix. For this second pass, CWH will iterate over the columns from left to right. In each column, all possible group formations will be determined and considered recursively. When the last column has been reached and all the possible group combinations have been considered the algorithm will end.

Whenever the number of remaining tiles hits 0 during the running of the algorithm, a solution has been found. The solution will be stored in a set and the function will return to its recursive call. Storing all the winning configurations found so far in a set (unique list) is required because there are certain configurations that CWH will enumerate twice. Figure 4 shows an example of such a configuration. This configuration could be built by placing runs of `123` for 3 different colors or placing the groups `111`, `222`, and `333`.

1	1	1	0
1	1	1	0
1	1	1	0

Figure 4: A configuration that could lead to a duplicate count. Configuration settings are $n = 4, k = 3, m = 1, h = 6$.

4.1.1 Pseudo-code

Algorithm 1 shows the pseudo-code of CWH. The algorithm can be split into two distinct parts, namely placing tiles as runs (lines 5-12) and placing tiles as groups (lines 15-20). The variable `placing` distinguishes on what iteration the algorithm is.

Like in the previous section, CWH will consider a configuration as a matrix. Variables `i` and `j` indicate the current position of CWH on this matrix. In every position, it is evaluated what the options are (lines 7-8 and 16-17), from which the options are explored with a recursive call (lines 10 and 19).

As soon as a full iteration of the matrix has concluded but the algorithm still has tiles to play (`hand_size` $\neq 0$), a different recursive call will be done (line 13). The positional variables will be reset to 0 and the `placing` variable will be flipped to the `GROUPS` state, indicating that the iteration for groups will now start.

Throughout this pseudo-code, several helper functions are used. The functions `determineRuns` and `determineGroups` determine all possible runs or groups of tiles that can be played at that current position and state of the configuration. The placing of a run should for example never exceed the maximum value a tile can take. Similarly, the size of a run or a group is also bound by the number of tiles that remain to be played. If that amount is 4 then we can also only place runs of 4, following the logic described in section 4.1. Possible group and run placement can also be bound by what tiles are already added to the configuration since we can only have m copies of the same tiles. The function `place` adds a run or group to a configuration.

Algorithm 1: Pseudo code of CWH() with the arguments (*configuration*, *hand_size*, $i = 0$, $j = 0$, *placing* = RUNS)

```
1  if $hand\_size = 0$ then
2      solutions.add($configuration$)
3  else
4      if $placing$ = RUNS then
5          while $j < max\_colours$ do
6              while $i < max\_value$ do
7                  for $run$ in determineRuns($config$, $i$, $j$, $hand\_size$, RUNS) do
8                      $new\_config$ = place($configuration$, $run$, $i$, $j$)
9                      CWH($new\_config$, $hand\_size - run.size$, $i$, $j$, RUNS)
10                 $i++$
11             $i = 0$, $j++$
12         CWH($configuration$, $hand\_size$, 0, 0, GROUPS)
13     else
14         while $i < max\_value$ do
15             for $group$ in determineGroups($config$, $i$, $hand\_size$, GROUPS) do
16                 $new\_config$ = place($configuration$, $run$, $i$, $j$)
17                 CWG($new\_config$, $hand\_size - group.size$, $i$, 0, GROUPS)
18             $i++$
```

5 Implementation

During the process of testing and gathering results, some improvements were made to the algorithm Count Winning Hands. This section discusses how some of the design choices and improvements were made during the development of the algorithm.

5.1 Programming language

CWH is developed in `Python 3.8`, but it should be possible to implement this in most other programming languages. The main motivation for choosing this language is the flexibility, readability, and ease of use it brings. It was not until later (see section 7) when some of the computational limits of `Python` were reached and we considered that more low-level programming languages like `c++` might have an edge for this implementation.

There are 4 external packages used in this implementation. The first two are `timeit` and `psutil` to gather some statistics about the runtime and memory usage respectively. The `itertools` package allows for some efficient array operations. This leaves `multiprocessing`, a package used to implement the multi-threading version of this algorithm, further discussed in section 6.

Furthermore, tests are (unless noted otherwise) run on an Intel Core i7 9700k with 16 GB of RAM. This is a processor with 8 logical cores and does not support hyper-threading.

5.2 Solution mapping

In section 4.1 we describe why all solutions are stored in a set until the algorithm is done. To manage the memory usage of this set, the solutions must be stored as efficiently as possible.

In our `Python` implementation, configurations are kept as nested lists because it is easy for CWH to work with. However, when a configuration is stored as a solution there is no longer a need to keep this nested list structure. By mapping the configuration to an integer, we can reduce the memory usage as our solution set grows larger.

Figure 5 visualizes an example of this mapping. Here, every row is read as a binary number. All these binary numbers are then concatenated to form one integer. This integer is then added to the solution set.

1	1	1	0
1	1	1	0
1	1	1	0

$\longrightarrow 010001000100 = 1092$

Figure 5: Visualisation of the solution map function. $n = 4, k = 3, m = 1, h = 3$.

To give an indication of what effect this will have, consider the example below. A random configuration is initialized in the `table` variable. Using a `getsizeof()` call on this variable returns a size of 88 bytes. Now we use our solution mapping function `hashTable()` on the configuration to map the configuration to an integer. This operation reduces its size to 40 bytes, a reduction of more than half. This is a significant improvement, especially considering how large solution sets can grow. This size can likely be reduced even further.

```
>>> table = [[2,1,1,0,1,1,1,0,1,1,1,0],
...           [0,1,1,1,1,1,2,0,1,1,1,0],
...           [1,1,1,1,1,2,1,0,1,1,1,0],
...           [1,1,1,2,1,1,1,0,1,1,1,0]]
>>> getsizeof(table)
88
>>> getsizeof(hashTable(table))
40
```

5.3 Parallel Processing

As runtimes started to increase towards multiple hours it became more desirable for the algorithm to make use of multiple threads. To implement multi-threading we need to distinguish a main problem and its sub-problems. Then, by assigning these sub-problems their thread, they can be processed in parallel instead of sequentially, resulting in a performance gain in terms of time.

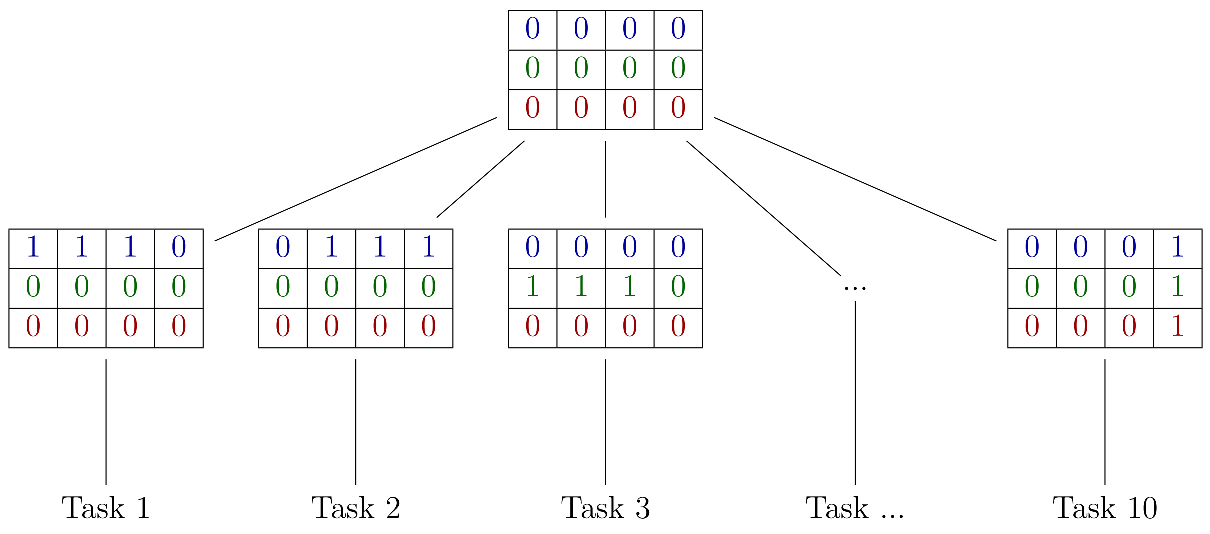


Figure 6: Visualisation of the Multi-threading implementation of CWH-parallel through a simple example.

With CWH, each recursive call made is essentially the algorithm splitting up the problem into smaller problems for the recursive call to figure out. Therefore, a method of implementing multi-threading would be to take all recursive calls at a recursion depth of level 1 as a list of tasks. This list of tasks can then be spread over any number of threads. Figure 6 shows a visualization of this process.

The performance gain for this implementation scales nearly linear with the number of cores assigned. Figure 7 shows this difference in performance with different configurations of 6 values, 4 colors, and 2 copies of each tile.

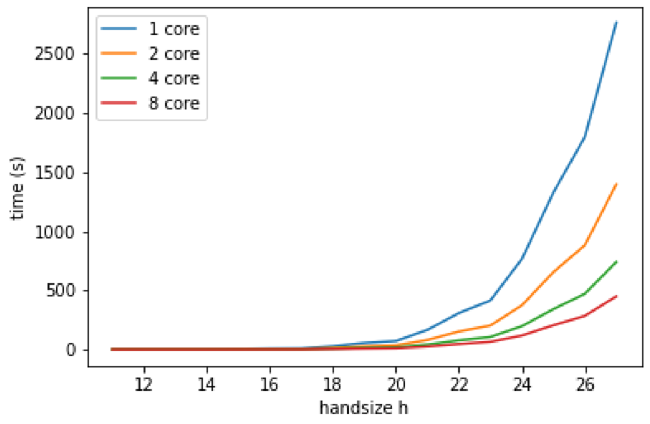


Figure 7: Multi-threading performance for $n = 6, k = 4, m = 2$. This implementation shows a nearly linear performance gain for each added core.

This implementation does have one drawback namely an increase in memory usage. The cause lies with each thread building its own solution set and not sharing these solutions with its parent or other threads. Meanwhile, other threads might hold the same solution. This is only resolved when a thread finishes and merges its solution set with that of its parent. The exact impact on memory is not clear and was hard to measure within Python and multiple threads that barely communicate. In an ideal situation, multiple threads can write their solutions to a shared solution set.

5.4 Knowing when to stop

During some of the initial runs it was noticeable that as the hand size h increases so did the runtime of the algorithm. The maximum runtime was reached for the maximum value of $h_{max} = n \times k \times m$. This seems excessive as for h_{max} there exists only one possible winning configuration. One would expect the problem to be relatively simple and the runtime therefore relatively low. In fact, we expect lower runtimes as h nears either 0 or h_{max} and a maximum runtime somewhere in between.

For CWH to show a tipping point in runtime as h reaches h_{max} the algorithm must know when to stop searching for solutions and prune. This can be achieved by checking if there still exists a certain best-case scenario at every iteration of the algorithm. This best-case scenario can be described as a scenario in which I can still play all the tiles currently in hand. Figure 8 shows how an example of how this could work in practice. In the first scenario, it is still possible to play the 5 tiles currently in hand. However, on the next iteration, there is no longer room to play these 5 tiles; the algorithm can stop.

$$\begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 \\ \hline X & X & X & X \\ \hline \end{array} \rightarrow \text{PRUNE}$$

Figure 8: Visualisation of the counting ahead optimization. $n = 4, k = 3, m = 1, h = 8$. In the first scenario, we still see an opportunity to play our 5 remaining tiles. In the second scenario, we no longer have this opportunity.

It should be noted that this "best"-case scenario can be specified in various degrees. The example in figure 8 is a relatively simple specification (and also the current implementation). A more strict version could also prune on the first scenario because on row 2 there is only space for 1 more tile and the algorithm will never place less than 3 tiles at once. However, specifying this scenario too extensively might in turn lead to an impact on performance that does not justify compared to a simpler specification.

Already, the simple implementation from figure 8, seems to have a notable negative impact on performance. Figure 9 shows the time taken to find all solutions per hand size h, for the configuration of 6 values, 4 colors, and 2 copies of each tile. It shows an increasing (growing with h) effort of finding solutions as opposed to not using this algorithm.

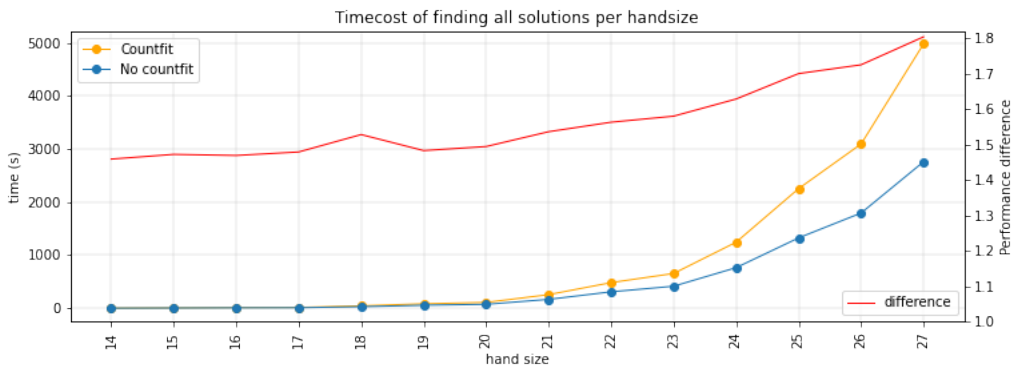


Figure 9: Performance of the countfit optimization showing an increasing effort as h grows. Configuration settings are: $n = 6, k = 4, m = 2$.

6 A backward approach

In this section, we discuss a different approach that was briefly explored in an attempt to tackle some large hand sizes near h_{max}. This algorithm starts under the assumption the player has all tiles (instead of none like with CWH). It continues to iterate over this set of tiles, leaving out a different combination of tiles every iteration and testing if the hand remains winning. The size of the combination that is left out is $h_{max} - h$ and is thereby dependent on the target hand size h. Figure 10 shows an example for a tile set with $h_{max} = 4 \times 3 \times 2 = 24$ and a target hand size of $h = 23$.

$$\begin{array}{|c|c|c|c|}\hline 1&2&2&2\\\hline 2&2&2&2\\\hline 2&2&2&2\\\hline\end{array}\ ,\ \begin{array}{|c|c|c|c|}\hline 2&1&2&2\\\hline 2&2&2&2\\\hline 2&2&2&2\\\hline\end{array}\ ,\ \ldots\ ,\ \begin{array}{|c|c|c|c|}\hline 2&2&2&2\\\hline 2&2&2&2\\\hline 2&2&2&1\\\hline\end{array}$$

Figure 10: Visualisation of the backward approach. Essentially this algorithm is repeatedly removing different tiles from the full set of tiles and checks if the configuration remains winning after. $n = 4, k = 3, m = 2, h = 23$.

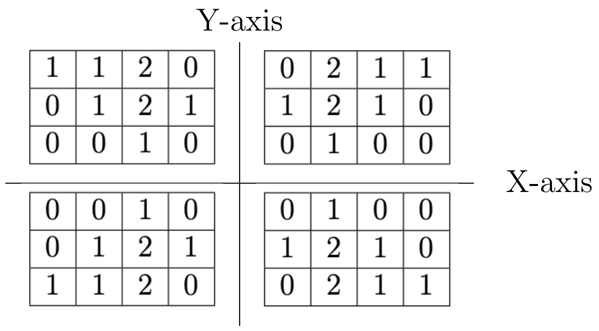
The implementation makes use of `c++` script developed by Frank W. Takes. It takes a list of configurations as input and determines for each of the given configurations whether it is winning or not.

By writing a `Python` wrapping script that generates all possible configurations, removes duplicates (section 6.1), and automatically feeds these to Takes' script, we create a program that essentially has the same outcome as CWH. The program has also surprisingly efficient memory usage because solutions are written to a file on disk. It is also highly scalable in terms of CPU processing. Parallelizing this script is simply calling Takes' script a multitude of times.

Although this program seemed promising at first, it soon became clear that the more h deviates from its maximal value h_{max} harder the runtime was growing. This implementation is practically brute-forcing to find its answer and we earlier established in section 4 that is a very naive approach with the number of total possibilities that exist. Later the optimization discussed in section 5.4 was found and it should make this method even less relevant.

6.1 Mirroring

Developing the backward approach brought to light the concept of mirroring solutions. This is a concept that has the potential to significantly reduce the number of solutions needed to compute in certain counting problems of Rummikub. The idea is that when a winning Rummikub solution is mirrored over the X-axis or Y-axis (or both), then a different winning solution will be found, see figure 6.1. Mirroring a configuration does not alter any connection between tiles, only the values of those tiles will change, which means the configuration will hold to the important rule that every tile is played in a series of minimally 3 tiles. This means if an algorithm finds a winning configuration it potentially could deduct up to 3 more winning configurations to add to its solution set. For the backward approach, this meant that for certain configurations, if we found a configuration that was winning its mirrors could be cleared as well and added to the solutions without checking. This mirroring concept has not found an implementation for the CWH algorithm.



1	1	2	0
0	1	2	1
0	0	1	0

0	2	1	1
1	2	1	0
0	1	0	0

0	0	1	0
0	1	2	1
1	1	2	0

0	1	0	0
1	2	1	0
0	2	1	1

Figure 11: By mirroring a solution over the X & Y axis other solutions are found.

7 Experiments

A method for counting winning Rummikub hands is described in section 4. Section 5 discusses a possible implementation of this algorithm and further mentions a couple of improvements. Most of these improvements, as well as the alternative approach from 6, have been developed during the processing of testing and running experiments to reach some set-out goals.

The first of these goals was to validate some of the results attained by van Rijn et al., 2015. Although this validation is not so much of a challenge computationally and thus reached relatively quickly, it is still a very important milestone. Both algorithms used were developed independently and if both come to the same results it gives confidence that these are correct.

Subsequently, was to gather some results that are more of a challenge to gather computationally. It is during this process that the bottlenecks were found and therefore also the source of most of the improvements and experiments from the previous sections.

In this section, both of these topics will be addressed and their respective results will be discussed.

7.1 Validating

In the work of van Rijn et al., 2015, two sets of results are given to compare against. The first set of results uses the same settings as those of the default Rummikub rules, namely 4 colors, 13 values, and 2 copies of each tile. Table 1 shows what is generated by our implementation.

Table 1: Number of winning Rummikub configurations with a hand of h tiles for a configuration size of 13 values, 4 colors, and 2 copies of each tile.

h	solutions
3	96
4	53
5	36
6	4,656
7	4,980
8	4,731
9	151,728
10	233,412
11	279,108
12	3,753,318
13	7,244,080
14	10,232,524
15	75,493,324

The first and most important thing to note from these results is that they correspond with those found by van Rijn et al., 2015. It indicates that the results both parties found seem to match up and are thereby somewhat meaningful.

Secondly, running the algorithm quickly becomes increasingly computationally expensive both in terms of time and in terms of space. Running for larger hand sizes h with these settings will require hardware with greater capabilities than those at hand.

Scaling down the problem, using smaller input, does make it possible to work out the number of winning situations for all hand sizes h. For this instance, the settings are still

4 colors and 2 copies of each tile, but the number of values is limited to 6 instead of 13. Figure 12 shows the results for these settings. Crucially the results for these settings again seem to match with that of van Rijn et al., 2015.

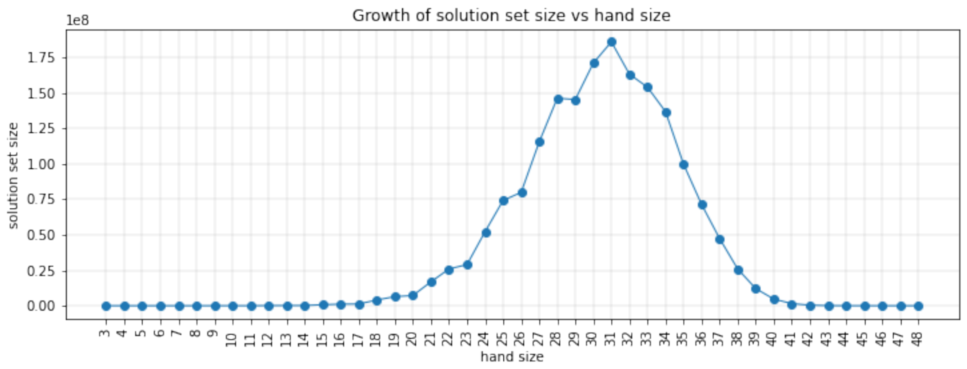


Figure 12: Solution set size vs the hand size h for 6 values, 4 colors, and 2 copies of each tile. The peak solution set size is found a $h = 31$.

This figure shows that the maximum solution set size is reached at 31, after which the solution sets start declining again. An effect to notice from this figure is that every for every 3rd increment of h ($h \bmod 3 = 0$) generally goes with a substantial increase of the solution set size. The effect is even more clear when the solution set size is plotted on a logarithmic scale, see figure 13. As the solution set grows to its maximum size, the effect seems to change. In figure 13, a decrease in the solution set size can be after the first few increments of 3. After a while, however, this decrease seems to turn into an increase. Such an increase is easier to read in figure 12, where the increase in solution size from $h = 27 \to h = 28$, is nearly as big as the increase from the 3rd increment: $h = 26 \to h = 27$.

A cause for this effect is likely linked to the minimal group and run size requirement, recall this is set to 3 for this thesis. Every 3rd increment of h opens the opportunity to create a different and separate set (not connected) to any of the other sets. The decrease in the solution set size can be after the first few increments of 3, which is also caused by this minimum size requirement. If a run/group size is smaller, there are more ways to play it. An example would be $h = 5$, which cannot be played in any groups, whereas $h = 3$ can be played in 4 different combinations.

Figure 13 also plots the number of different partitions (of size 3,4 or 5) that can be formed given a hand size h plotted. For example, $h = 9$ can be partitioned in $3+3+3$ but also in $4+5$. Unsurprisingly, this number plotted over h shows that similar periodic incrementing (every 3rd h) as seen with the solution set size, confirming the aforementioned cause.

A computational wall is quickly hit for an instance of 13 values but counting all solutions for 6 values seems to be doable in a reasonable time. Consequently one might

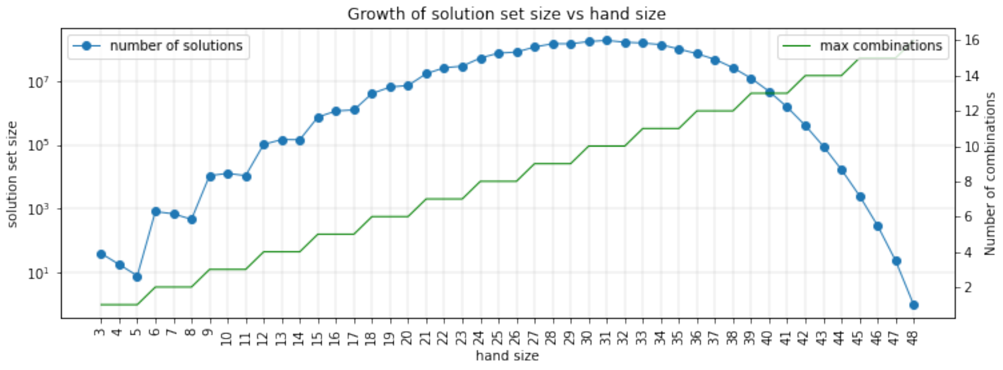


Figure 13: Solution set size vs the hand size h for 6 values, 4 colors, and 2 copies of each tile. Notice the significant increase in solution every 3^{rd} increment of h for the first 30 hand sizes. This behavior is in line with the maximum number of combinations of 3,4,5 that h can consist of.

wonder how this would fare for an instance of 7 values. The next section will explore this scenario.

7.2 Expanding

Most of the improvements mentioned in 5 and 6 originate from one of the main goals of this thesis: finding all solution sizes for a Rummikub instance with 7 values, 4 colors, and 2 copies of each. Although this set of input might only differ slightly from the one discussed in the previous section (values = 6, section 7.1), the number of solutions increases immensely and made it computationally challenging to figure out. Figure 14 show the results achieved so far.

While the results are incomplete, the biggest solution size has been found (at $h = 37$). For comparison, the curves for $n = 5$ and $n = 6$ are also plotted. The resemblance between the 3 curves is noticeable and could opportune to start predicting unknown values. For example, the maximum value of each of these curves is at $62,5\%$, $64,6\%$, and $66,1\%$ of h_{max} for respectively n is 5,6 and 7. Extrapolating from these values it would seem plausible that $n = 8$ would have its maximum solution set at around 67% or 68% which would translate into $h = 43$. However, more research is required to validate this idea.

These results are gathered using the two different techniques parallel-CWH and the backward approach explained in section 5 and 6 respectively. It is also no longer possible to use home PCs for these runs since the hardware is no longer capable of handling these large solution sizes. Because of the difference in nature of the above-mentioned algorithm, one requiring more RAM and the other preferring more cores, two different servers are selected. For parallel-CWH, a server with 64 cores and 1TB RAM is used. The algorithm will only use 16 of those 64 cores. For the backward approach, a server

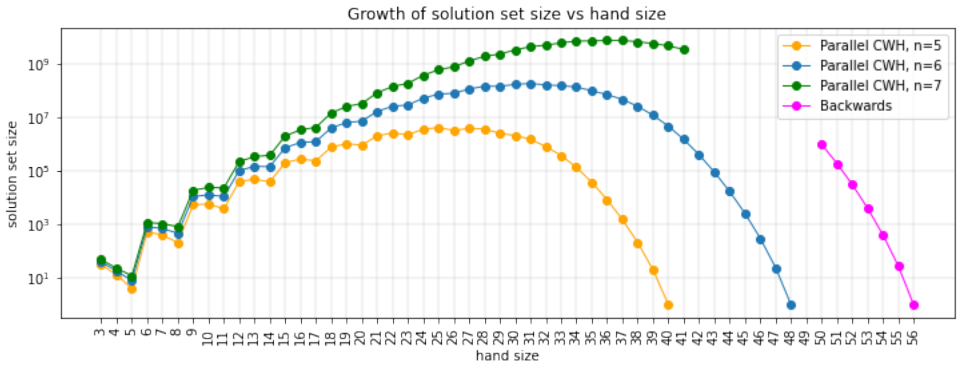


Figure 14: Solution set size vs the hand size h for different n values, 4 colors, and 2 copies of each tile. The different values all follow a similar curve. The gap in the curve $n = 7$ represents the solutions that have yet to be found.

with 256 cores is used of which only 64 are used. These servers are shared with other users and therefore they cannot be used for benchmarking. Stability was also an issue because of this, with unplanned server restarts or the server running out of memory when simultaneously running different resource-intensive programs.

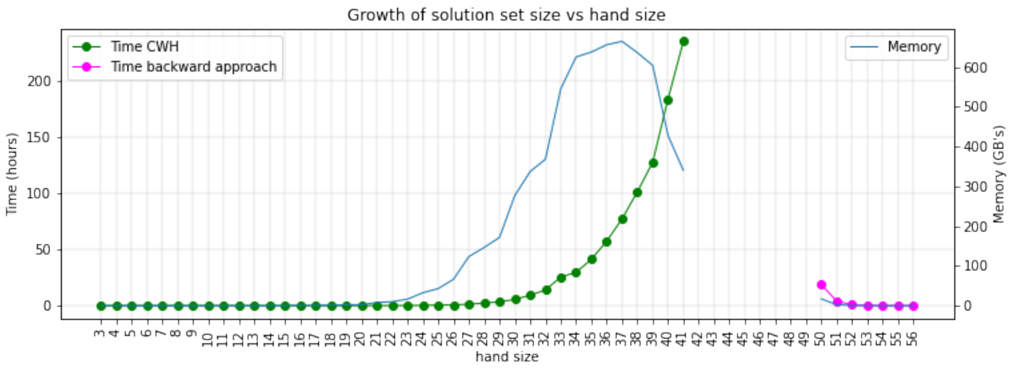


Figure 15: Solution set size vs the hand size h for 6 values, 4 colors, and 2 copies of each tile. The largest solution set size has been found. The gap in the curves represents the solutions that have yet to be found.

With parallel-CWH, results have been gathered up until a hand size h of 41. While time and memory usage has been measured, see figure 15, both can only be used as a rough measure. Run-time is could be impacted by other users using the server and the size of this impact is unknown. Furthermore, the memory usage from the different threads was at times incorrectly added up, and actual peak usage numbers are expected to be higher. Nevertheless, both still form a solid indication and are worth mentioning, to get an idea of where performance is at. Combined peak memory usage for this instance is seen well over 600 GB. Run times beyond $h = 41$ are also going into the weeks, with $h = 40$ taking over 7 days and $h = 41$ nearing 10 days. Since this parallel-CWH implementation does not have the counting ahead implementation (section 8), the growth in time will also not stop. At this point, server stability became an issue with unplanned restarts or the algorithm being killed when simultaneous with other resource-intensive programs. Therefore no further attempts were made to use this algorithm. With the backward approach, it was possible to find the solution set sizes of hand 56 to 50. It might still be possible to find the solution set size for $h = 49$, with this method but here server stability also became an issue similarly.

8 Conclusions and Further Research

Throughout this thesis, we have been busy with several aspects of this one problem of counting winning Rummikub hands. To tackle this problem an algorithm (CWH) is proposed and explained.

Crucially when put to the test, this algorithm seems to reproduce the same results found by van Rijn et al., 2015 earlier. This gives some confidence that the algorithm and the found results are correct.

To expand on these results and find for what size input this can be calculated with current hardware, we have attempted to push the algorithm for a larger input size. Since this input turns out to be challenging, various optimizations have been made to method to improve performance, each again thoroughly explained. Nevertheless, we were not able to find the solution for all Rummikub hand sizes of that input. While this might seem that the results are thereby somewhat incomplete, we have achieved success in finding a computational limit for this implementation.

The runtime of our implementation was long, however more importantly it was no longer scalable because of the excessive memory usage. The need to keep a solution set with all current solutions found to avoid registering duplicate solutions forms the bottleneck for this algorithm. A possibility of addressing this would be to improve the storage and access of the solution set, also considering a multi-threaded implementation. Another arguably more ideal solution would be to find an algorithm that does not find duplicate solutions in the first place since that would eliminate the need for keeping an ever-growing solution set. Either way, there is still a lot of potential for improvement for this algorithm to allow for more efficient operation, both in terms of implementation and logic.

In this thesis, we mostly considered variations of problems by changing the values n from the default Rummikub tile set. Other parameters like the number of colors/copies of each tile & minimal group/run size remained untouched. It could be interesting to see how these parameters would impact the problem of counting winning Rummikub hands.

Perhaps most interesting for game theory, would be to consider this problem or the game of Rummikub in general with adversaries. Here, the leading question could be if it would be possible to build a model or strategy and what this would look like.

This is also what we hope this research will eventually contribute to, through means of finding the probability of winning. The game of Rummikub includes decision-making since in a turn, a player sometimes has the option to either draw a tile or play some of its tiles. It remains to be seen if this decision-making can be (situationally) influenced by knowledge of the odds that a particular hand could be winning.

References

den Hertog, D., & Hulshof, P. B. (2006). Solving rummikub problems by integer linear programming. *Comput. J.*, *49*(6), 665–669.

Gulin, M. (2019). *Solving rummikub with computational power.*

Kotnik, C., & Kalita, J. K. (2003). The significance of temporal-difference learning in self-play training td-rummy versus evo-rummy. *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003)*, 369–375.

Rummikub: How to play. (n.d.). https://rummikub.com/rules/ (accessed: 09.10.2022)

van Rijn, J. N., Takes, F. W., & Vis, J. K. (2015). The complexity of rummikub problems. *Proceedings of the 27th Benelux Conference on Artificial Intelligence (BNAIC 2015).*

Appendix

Table 2: Number of winning Rummikub configurations with a hand of h tiles for a configuration size of 6 values, 4 colors, and 2 copies of each tile.

h	solutions	h	solutions	h	solutions
3	40	19	6,433,240	35	99,759,056
4	18	20	7,220,872	36	71,349,974
5	8	21	16,853,400	37	47,317,392
6	820	22	25,910,748	38	25,908,596
7	696	23	29,036,248	39	12,096,784
8	467	24	52,234,799	40	4,708,057
9	10,872	25	74,258,224	41	1,509,768
10	12,816	26	79,916,256	42	404,184
11	10,896	27	115,475,104	43	90,720
12	103,340	28	146,292,716	44	16,974
13	146,760	29	145,376,712	45	2,576
14	144,856	30	171,072,496	46	300
15	738,648	31	186,041,792	47	24
16	1,150,642	32	163,221,584	48	1
17	1,240,616	33	154,071,432		
18	4,042,944	34	136,832,104		

Table 3: Number of winning Rummikub configurations with a hand of h tiles for a configuration size of 7 values, 4 colors, and 2 copies of each tile. Note that the solutions with a * are not found yet.

h	solutions
3	48
4	23
5	12
6	1,176
7	1,068
8	816
9	18,912
10	24,004
11	23,088
12	222,549
13	342,624
14	382,880
15	2,023,488
16	3,441,477
17	4,225,324
18	14,567,450
19	25,528,004
20	32,957,292
21	83,299,676
21	83,299,676
22	142,777,052
23	186,805,904
24	374,025,439
25	603,930,676
26	775,426,266
27	1,290,486,380
28	1,910,673,477
29	2,338,326,204
30	3,318,889,954
31	4,410,877,676
32	5,002,983,709
33	6,104,467,324
34	7,132,482,924
35	7,277,229,084
36	7,572,253,716
37	7,575,982,124
38	6,703,667,162
39	5,792,305,988
40	4,763,463,909
41	3,468,285,700
42	*
43	*
44	*
45	*
46	*
47	*
48	*
49	*
50	985674
51	189672
52	30681
53	4032
54	406
55	28
56	1